RICE UNIVERSITY

# Task and Motion Planning for Mobile Manipulators

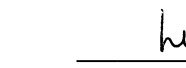
by

**Ioan Alexandru Şucan**

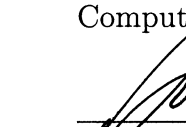
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

**Doctor of Philosophy**

APPROVED, THESIS COMMITTEE:



Dr. Lydia E. Kavraki, Professor, Chair
Computer Science, Rice University



Dr. James McLurkin, Assistant Professor
Computer Science, Rice University

Dr. Marcia K. O'Malley, Associate Professor
Mechanical Engineering, Rice University

Dr. Mark Moll, Adjunct Assistant Professor
Computer Science, Rice University

HOUSTON, TEXAS
AUGUST 2011

## Abstract

### Task and Motion Planning for Mobile Manipulators

by Ioan Alexandru Şucan

This thesis introduces new concepts and algorithms that can be used to solve the *simultaneous task and motion planning* (STAMP) problem. Given a set of actions a robot could perform, the STAMP problem asks for a sequence of actions that takes the robot to its goal and for motion plans that correspond to the actions in that sequence. This thesis shows how to solve the STAMP problem more efficiently and obtain more robust solutions, when compared to previous work. A solution to the STAMP problem is a prerequisite for most operations complex robots such as mobile manipulators are asked to perform. Solving the STAMP problem efficiently thus expands the range of capabilities for mobile manipulators, and the increased robustness of computed solutions can improve safety.

A basic sub-problem of the STAMP problem is motion planning. This thesis generalizes KPIECE, a sampling-based motion planning algorithm designed specifically for planning in high-dimensional spaces. KPIECE offers computational advantages by employing projections from the searched space to lower-dimensional Euclidean spaces for estimating exploration coverage. This thesis further develops the original KPIECE algorithm by introducing a means to automatically generate projections to lower-dimensional Euclidean spaces. KPIECE and other state-of-the-art algorithms are implemented as part the Open Motion Planning Library (OMPL), and the practical applicability of KPIECE and OMPL is demonstrated on the PR2 hardware platform.

To solve the STAMP problem, this thesis introduces the concept of a task motion multigraph (TMM), a data structure that can express the ability of mobile manipulators to perform specific tasks using different hardware components. The choice of hardware components determines the state space for motion planning. An algorithm that prioritizes the state spaces for motion planning using TMMs is presented and evaluated. Experimental results show that planning times are reduced by a factor of up to six and solution paths are shortened by a factor of up to four, when considering the available planning options. Finally, an algorithm that considers uncertainty at the task planning level based on generating Markov Decision Process (MDP) problems from TMMs is introduced.

## Acknowledgements

First of all, I would like to thank my adviser, Dr. Lydia Kavraki, for her guidance and help throughout my graduate studies. I would also like to thank to my thesis committee members, Dr. Marcia O'Malley, Dr. James McLurkin and Dr. Mark Moll, as well as the members of the Physical and Biological Computing Group, for helpful discussions and for insightful comments.

I would like to thank Willow Garage for providing hardware support for many of the experimental results included in this thesis. I would also like to thank Mark Moll for his contributions to OMPL. Many thanks go to Marius Şucan for designing most of the images this document includes.

This work was supported in part by NSF IIS 0713623, NSF DUE 0920721 and Rice University funds. The experiments were run on equipment obtained through NSF grant number CNS-0421109 in partnership with Rice University, AMD and Cray, and through NSF grant number EIA-0216467 in partnership with Rice University, Sun Microsystems, and Sigma Solutions, Inc.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

One of the overarching goals of robotics is to create robotic devices that can take as input high-level specifications of tasks and execute them without requiring low-level instructions on how to implement that execution [1]. This is a difficult endeavor and requires solving a broad range of problems. This thesis relates to the problems of motion planning and of task planning, which typically need to be solved for a robot to move in its environment. Loosely stated, *motion planning* is the problem of finding the set of inputs to the robot's actuators – a *motion plan* – such that the robot moves from an initial to a final position while respecting a set of constraints (e.g., collision avoidance) [1], and *task planning* is the problem of finding the sequence of high-level actions – a *task plan* – the robot needs to take in order to achieve its goal [2] (e.g., reach for a tool and then bring it to a user). The execution of the high-level actions often requires corresponding motion plans.

Robots for planetary exploration, museum tour guides, search and rescue robots,

service robots and robots in surgery are just a few of the many examples of robotics applications that need task planning [2] as well as motion planning [3, 4]. In general, the motion planning problem can be viewed as search in high-dimensional continuous spaces. As such, its applications have expanded to other domains such as graphics, computational biology and verification [5–8]. For simplicity in presentation, this thesis will present developments in motion planning in the context of robotic systems. However, most of the contributions to motion planning described herein apply to other fields as well.

The task planning problem and the motion planning problem are closely related. For example, if an action selected at task planning level cannot be executed due to the inability of a motion planner to find the corresponding actuator inputs that implement that action, this information needs to be passed to the task planner so that the same action is not selected repeatedly. Furthermore, when a motion planner computes corresponding motion plans for two consecutive actions, it should ensure that the motion plan computed for the first action does not prohibit the computation of a motion plan for the second action. Because of the necessary information exchange, the task planning and the motion planning problems are often solved simultaneously. In this thesis, *simultaneous task and motion planning* (STAMP) refers to the problem of simultaneously selecting a task plan from a set of available possibilities and computing a corresponding set of motion plans. A formal definition of this problem is given later. A solution to the STAMP problem requires *(1)* identifying a task plan that takes the robot to its goal and *(2)* simultaneously computing the necessary motion plans for that task plan. The STAMP problem as defined in this thesis does not include

generating possible sequences of actions, which is often included in task planning as considered in artificial intelligence [2]. Previous work has addressed the task and motion planning problem (e.g., [9–15]), but more work is needed towards developing fast algorithms that can be used in reactive systems. The contributions this thesis makes to task and motion planning are intended primarily for complex robots that can perform specified tasks using different sets of their hardware components. Such robots are typically mobile manipulators – robots capable of both locomotion in the environment and manipulation of objects in the environment.

This thesis is structured in roughly two parts: the first part presents algorithms, techniques and tools for solving the motion planning problem, and the second part presents techniques for solving the STAMP problem. The second part of the thesis builds upon developments introduced in the first part. While individual contributions included in this thesis are often applicable in a variety of contexts, the thesis as a whole develops planning techniques adequate for task and motion planning for mobile manipulation platforms.

## 1.1 Definition of Problems

This thesis addresses two problems: the motion planning problem, defined in Section 1.1.1, and the simultaneous task and motion planning problem, defined in Section 1.1.2.

### 1.1.1 Motion Planning

In the simplest form, motion planning is the problem of finding a continuous path that connects a given start state to a given goal state (see Figure 1.1), under some specified set of constraints [1]. The specified constraints may include collision avoidance, maintaining orientation of certain robot parts, bounds in velocity, bounds in acceleration, etc. Depending on the type of constraints, two versions of the motion planning problem are distinguished: planning under geometric constraints (sometimes called "path planning" [3]) and planning under differential constraints (this version of the problem includes "kinodynamic motion planning" [3]).


**Figure 1.1:** A generic representation of the motion planning problem.

**Planning under Geometric Constraints** An instance of the *Motion Planning Problem under Geometric Constraints* can be represented as a tuple ($\mathcal{X}$, $isValid$, $S$, $G$) where:

- $\mathcal{X}$ is a set that represents the *state space* of the robotic system (sometimes called the configuration space [1]). A point $x \in \mathcal{X}$ fully characterizes the state of the system planning is performed for.
- $isValid : \mathcal{X} \rightarrow \{\top, \bot\}$ is a function that decides whether a state is valid or not.

This function effectively separates the state space into two disjoint subsets: the set of valid states $\mathcal{X}_{valid} = \{x \in \mathcal{X} \mid isValid(x) = \top\}$ and the set of invalid states $\mathcal{X}_{inv} = \mathcal{X} \setminus \mathcal{X}_{valid}$.

- $S \subseteq \mathcal{X}$, $S \cap \mathcal{X}_{valid} \neq \emptyset$, is the set of possible start states.
- $G \subseteq \mathcal{X}$, $G \cap \mathcal{X}_{valid} \neq \emptyset$, is the set of possible goal states.

This version of the problem typically assumes robots can move instantaneously in any direction – only geometric constraints such as collision avoidance are considered. This assumption is reasonable for certain robotics applications, if for example there exists a controller capable of following a geometrically computed path. However, many practical applications require accounting for dynamic constraints such as bounded torques, friction, etc.

**Planning under Differential Constraints** An instance of the *Motion Planning Problem under Differential Constraints* can be represented as a tuple ($\mathcal{X}$, $\mathcal{U}$, *propagate*, $isValid$, $S$, $G$) where:

- $\mathcal{X}$, $isValid$, $S$ and $G$ are the same as for planning under geometric constraints, with the added requirement that $\mathcal{X}$ is a differentiable manifold.
- $\mathcal{U}$ is a set representing the *control space* for the robotic system.
- *propagate* : $\mathcal{X} \times \mathcal{U} \rightarrow Tg\mathcal{X}$ models the evolution of the robotic system as controls are applied, where $Tg\mathcal{X}$ is the tangent bundle of $\mathcal{X}$. *propagate* is

usually represented as a system of differential equations or it can be modeled using a simulator (e.g., [16]).

### Solution to the Motion Planning Problem

A solution to the motion planning problem is a continuous path $p : [0, T] \rightarrow \mathcal{X}$, for some $T \in [0, \infty)$, such that $p(0) \in S$, $p(T) \in G$, $(t \in [0, T]) \rightarrow (isValid(p(t)) = \top)$.

The representation of $p$ depends on the version of the motion planning problem being considered. If planning solely under geometric constraints, $p$ can be represented as a finite sequence of way-points: $p = (x_0, x_1, \ldots, x_n)$, such that $x_0 \in S$, $x_n \in G$. In this case it is assumed that a means of interpolating between states is known. This means of interpolation needs to consider the topology of $\mathcal{X}$ and allow generating the states on any motion segment between $x$ and $y$, for $x, y \in \mathcal{X}$ (from this point onwards we use the notation $\overline{x, y}$ to denote a motion segment). The validity of $p$ could then be written as $\forall i \in \{1, \ldots, n\}$ $((x \in \overline{x_{i-1}, x_i}) \rightarrow (isValid(x) = \top))$.

If planning under differential constraints, $p$ can be reconstructed from the robot's initial state, a finite sequence of controls to be passed to the *propagate* function and a corresponding sequence of durations to apply the controls for. A possible representation is then $p = (x_0, (u_0, \ldots, u_{n-1}), (t_0, \ldots, t_{n-1}))$; $x_i$ can be constructed from $x_{i-1}$ by applying *propagate* with the control input $u_{i-1}$ for the duration $t_{i-1}$; for a correct solution, $x_n$ has to be an element of $G$.

### 1.1.2 Simultaneous Task and Motion Planning

Practical applications of robotics require solving problems that are more complex than computing motion plans between two given states. For example, interaction with physical objects in the robot's surroundings is typically required. Such interaction often requires a robot to plan motions towards objects of interest, achieve contact, perhaps plan other motions that transport the object of interest and finally break contact. Such sequences of actions constitute a task plan. In the simplest sense, task and motion planning is the problem of finding the discrete, finite, sequence of actions a robot needs to perform in order to achieve its goal, and at the same time compute motion plans for corresponding actions. The set of possible sequences of actions is typically encoded as a graph such as the one shown in Figure 1.2. These graphs – the *task graphs* – can be specified explicitly (e.g., [17]) but are most often specified implicitly using compact representations such as Linear Temporal Logic (LTL) [18] or Stanford Research Institute Problem Solver (STRIPS)-like languages [2] (e.g., Planning Domain Definition Language (PDDL) [19]).

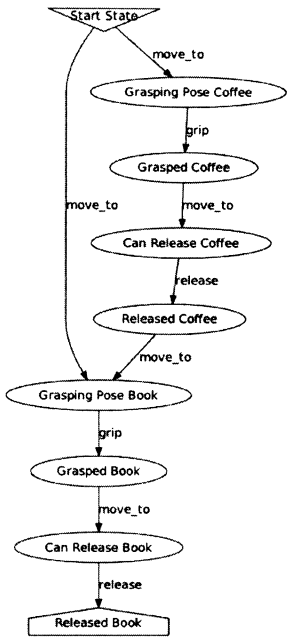



**Figure 1.2:** Example task graph.

This thesis uses a representation of task graphs that can accommodate all representations the author has encountered in the literature. This representation is described next. The notation is as follows:

- Let $\mathcal{X}$ be the state space of the robotic system a task graph is specified for.
- Let $\mathcal{A}$ be a finite set of atomic propositions. These atomic propositions represent properties of the system that can be true or false.
- Let $\mathcal{R}$ be a finite set of regions. Each region $R \in \mathcal{R}$ is in fact a set of states $R \subset \mathcal{X}$ (a subset of the state space).
- Let $\mathcal{T}$ be a finite set of actions. What these actions are depends on the representation of the task (STRIPS-like, LTL, etc). When motion planning is required as part of performing an action, the instance of the motion planning problem is associated to that action as well. In the case of a robotic arm, a typical example is $\mathcal{T} = \{\texttt{grip}, \texttt{release}, \texttt{plan}\}$; these actions correspond to closing the arm's end effector, opening the arm's end effector and moving the arm between two states, along a planned path.

**Representing Task Graphs** A task graph is a directed acyclic graph $G = (V, E)$, where:

- $V \subseteq \mathcal{R} \times 2^{\mathcal{A}}$, i.e., a vertex is identified by a region and a set of atomic propositions, where $2^{\mathcal{A}}$ denotes the power set of $\mathcal{A}$. For $v = (R, A) \in V$, $R \in \mathcal{R}, A \in 2^{\mathcal{A}}$. $R$ is called the region of vertex $v$ and $A$ is the set of propositions that are true at vertex $v$. The vertex $v$ encodes the task state of the robotic system. To refer to the region – the subset of $\mathcal{X}$ – that corresponds to a vertex $v$, we use the notation $Q(v)$.

- $E \subset V \times V$ such that $\forall v \in V, (v, v) \notin E$. The existence of an edge $(v_1, v_2) \in E$ implies there exists a potential means of arriving to $v_2$ from $v_1$ (an action in $\mathcal{T}$).
- $root \in V$ is the vertex designated as the starting task state for the robot. Typically this vertex has no parents and $Q(root)$ has one single element.
- $F \subseteq V$, $F \neq \emptyset$ is a set of vertices designated as goals of the task planner.

**Types of Actions** For an edge $e = (v_1, v_2)$, $v_1 = (R_1, A_1)$, $v_2 = (R_2, A_2)$, either $A_1 \neq A_2$ or $R_1 \neq R_2$. If $A_1 = A_2$ and $R_1 = R_2$ then the action would be a no-op and need not be included in the task graph.

If the only change performed by the edge's action is changing values of atomic propositions ($R_1 = R_2$, $A_1 \neq A_2$) the action is called an *observation* action. We treat observation actions differently, in the sense that the robot needs to physically reach the region corresponding to the action's starting vertex in order to continue its operations: it needs to observe its environment and make a decision. This is typically necessary for practical robotics applications. Actions that are not observation actions are called *sequence* actions. A sequence action between $v_1 = (R_1, A_1)$ and $v_2 = (R_2, A_2)$ implies changing the state of the robotic system from $x \in R_1$ to $x' \in R_2$ and potentially also updating the set of true atomic propositions.

**Simultaneous Task and Motion Planning** Given a task graph $G = (V, E)$, the *simultaneous task and motion planning* (STAMP) problem requires finding an ordered sequence of edges $P = \{e_1, \ldots, e_k\}, P \subseteq E$ such that if $e_i = (v_{i,a}, v_{i,b})$, we always have $v_{i,b} = v_{i+1,a}$, $v_{1,a} = root$ and $v_{k,b} \in F$. Furthermore, for each edge $e_i$, a corresponding

motion plan $m_i \subset \mathcal{X}_{valid}$ needs to be computed, between the states $x_{i,a} \in Q(v_{i,a})$, $x_{i,b} \in Q(v_{i,b})$ (as in Section 1.1.1). It is assumed that given two consecutive motion plans $m_i$, $m_{i+1}$, it is possible they can be *connected*: there exists a means to control the robotic system from $x_{i,b} \in Q(v_{i,b})$ to $x_{i+1,a} \in Q(v_{i,b})$ $(= Q(v_{i+1,a}))$. The difficulty of the STAMP problem lies in finding a sequence of actions for which corresponding motion plans can be computed and connected.

Since *observation* actions require the robot to actually reach specific regions in the state space, a complete course of action cannot be fully decided in advance. The algorithms presented in this thesis only operate on *sequence* actions. When an observation action needs to be taken, its starting vertex is designated as a goal. If the robot reaches a goal that is in fact the starting vertex of an observation action, the observation can then be performed. Using the result of the observation, sequence actions following the observation action can be then considered. In this manner, algorithms that only operate on *sequence* actions can be applied in real situations, when observations need to be made. For the remainder of the thesis we make the assumption that only sequence actions are included in task graphs.

## 1.2 Contributions

This thesis presents algorithmic contributions applicable towards solving two problems: the motion planning problem and the STAMP problem. The contributions are as follows:

- The KPIECE algorithm for planning under differential constraints is generalized and a method for generating random projections to lower-dimensional Euclidean spaces is described. The use of random projections enables KPIECE and similar algorithms (such as SBL [20] and EST [21]) to efficiently estimate the coverage of a robot's state space by considering discretizations of lower-dimensional Euclidean spaces. This improvement leads to computational advantages over previous work and increases ease of use by reducing required user input.

- A number of state-of-the-art sampling-based algorithms, including KPIECE, are implemented as part of a free software library called OMPL (The Open Motion Planning Library). OMPL is an easy to use, efficient library, useful in both academic and industrial settings. The practical applicability of OMPL and of the algorithms it includes are demonstrated through their integration with perception on the PR2 hardware platform (Personal Robot 2, from Willow Garage), as part of ROS [22]. Multiple versions of the KPIECE algorithm are included in OMPL: in addition to the version for planning under differential constraints, versions for planning under geometric constraints are also developed. Computational advantages over previous work are observed when planning with geometric constraints as well.

- The concept of a task motion multigraph (TMM) is developed. TMMs can be used to represent the available motion planning options for complex robotic systems such as mobile manipulators. This thesis shows how to encode a given task as a TMM.

- An algorithm that solves the STAMP problem using TMMs is presented and evaluated. The algorithm makes use of information from the TMM to prioritize the spaces for which motion plans are computed. Experimental results for office-like environments show that planning times can be reduced by as much as a factor of six and solution paths can be shortened by a factor of four, when considering the available planning options. The reduced computation time and improved quality of solutions can extend the applicability of motion planning for mobile manipulators to more complex tasks.

- An algorithm that considers uncertainty at the task planning level based on generating MDP (Markov Decision Process) problems from TMMs is presented and evaluated.

The contributions this thesis brings enable complex robotic systems such as mobile manipulators to efficiently and robustly solve the STAMP problem.

Algorithmic developments this thesis introduces for solving the motion planning problem enable fast computation of individual motion plans, while the concept of a TMM enables the exchange of information between the selection of possible task plans and the computation of motion plans, such that the STAMP problem can be solved up to six times faster than in previous work. TMMs also enable the consideration of uncertainty at task planning level, leading to robust solutions. The combination of speedup in computation and robustness of solutions makes the algorithms presented in this thesis applicable in practical scenarios.

## 1.3 Organization of the Thesis

In the first part of the thesis, Chapters 3 and 4 present algorithms, techniques and tools for solving the motion planning problem. Chapter 3 shows an improved version of the KPIECE algorithm that uses random projections, Chapter 4 describes OMPL and shows applications of KPIECE and OMPL on the PR2 hardware platform.

In the second part of the thesis, building upon the developments introduced in the first part, Chapters 5 and 6 present techniques for simultaneous task and motion planning. The concept of a task motion multigraph (TMM) is introduced in Chapter 5 and Chapter 6 shows how to account for uncertainty using TMMs.

# Chapter 2

# Background

## 2.1 Motion Planning

Two decades ago, the focus in motion planning was on *planning under geometric constraints.* This problem is sometimes referred to as the piano movers' problem, or in 2D, the sofa movers' problem, and it was the subject of extensive research [23–25]. A number of complete algorithms were developed for various cases of the problem and it was eventually shown to be PSPACE-complete [26, 27]. The developed algorithms are typically difficult to implement and computationally prohibitive by today's standards. Techniques such as cell decomposition methods and potential fields [1, 3, 4] were studied as well, but few were successful at solving problems where the state space is high-dimensional [28].

In addition to geometric constraints, planning for real robotic systems requires accounting for dynamic constraints (e.g., friction, gravity, limits in forces). In general it

is not known if this version of the problem, *planning under differential constraints*, is even decidable [29]. However, for the simplified case of a point mass robot, a polynomial algorithm exists [30]; the reconfiguration of modular robots under kinodynamic constraints is possible in $\Theta(\sqrt{n})$ time under certain assumptions [31].

The proven difficulty of planning under geometric constraints and the need to consider even more complex versions of the problem, such as planning under differential constraints, pushed the research in motion planning towards techniques with weaker completeness guarantees [3,4]. There are multiple directions of research that exhibit weaker completeness guarantees. This thesis relies on one of these directions, namely sampling-based motion planning, a direction in which promising results have been shown for planning in high-dimensional systems with complex dynamics [3,4,32,33].

**Sampling-based Motion Planning** Much of the recent progress in motion planning is attributed to the development of sampling-based algorithms [3,4]. Sampling-based motion planning algorithms relinquish completeness in favor of probabilistic completeness [34, 35]. Given sufficient time, a probabilistically complete algorithm will eventually find a solution with probability 1, if one exists. However, if no solution exists, the algorithm will not terminate.

One of the most influential algorithms in sampling-based motion planning was Probabilistic Roadmap Method (PRM) [36–38]. This method provided a coherent framework for many earlier works that used sampling and opened new directions for research. The core idea of PRM is to approximate the connectivity of the valid part state space, $\mathcal{X}_{valid}$. The approximation is represented as a graph called the roadmap.

The vertices of the roadmap are states sampled from $\mathcal{X}_{valid}$. For every vertex in the roadmap, connections to some $k$-nearest neighboring vertices are considered. If the motion of the robot between two vertices does not cross $\mathcal{X}_{inv}$, the corresponding edge is added to the roadmap. As more vertices and more edges are added, the roadmap data structure approximates $X_{valid}$ more accurately. When a user-specified query needs to be solved, the motion planning problem can be reduced to a graph search problem by connecting the start and goal states to the roadmap. A graphical representation of a roadmap is shown in Figure 2.1.

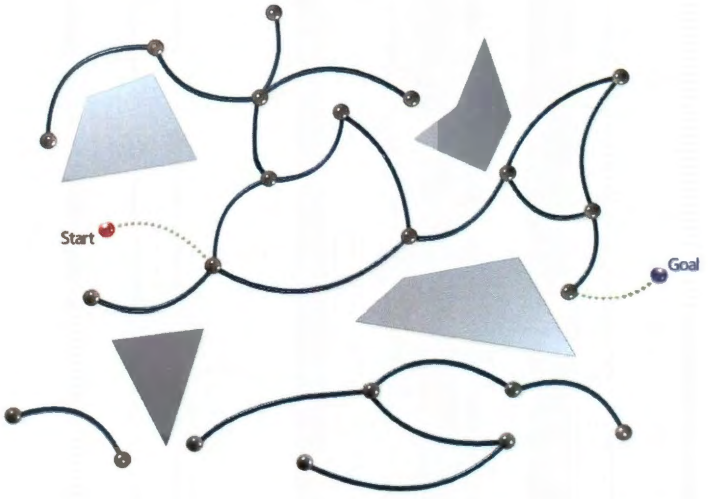



**Figure 2.1:** Graphical representation of a roadmap in a two-dimensional state space.

PRM inspired many other sampling-based algorithms [20,21,39–56]. Among these, a notable class is that of tree-based planners. As the name suggests, tree-based planners grow a tree in the state space of the robotic system. The initial tree consists of the robot's starting state. Newly sampled states are connected to some already

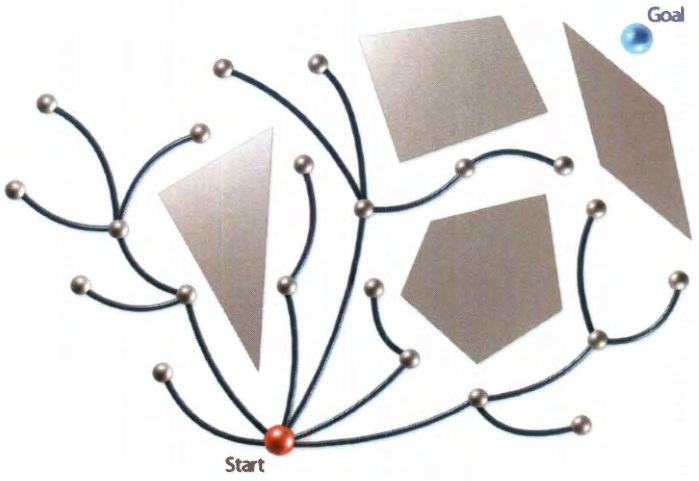



**Figure 2.2:** Graphical representation of a tree of motions in a two-dimensional state space.

existing state in the tree [57].

Tree-based motion planners are appropriate for planning under differential constraints because implementations that only use forward propagation are possible. If backward propagation is available, more efficient bi-directional tree planners can also be used, but in this case the steering problem [58, 59] needs to be solved as well. A further reason to use tree-based planners is that they explicitly consider the problem to be solved – the tree is grown from the start state towards the goal state – which typically allows finding a solution faster than approximating $\mathcal{X}_{valid}$ entirely. A representation of a tree of motions is shown in Figure 2.2. There are two fundamental issues tree planners must consider in their expansion:

*1) In which parts of the tree expansion should continue*: There are various ways to guide the tree expansion (e.g., [21,41–43,54,60–63]). Rapidly-exploring Random Trees

(RRT) expand from states closest to randomly produced states [42, 60], Expansive Space Trees (EST) and Single-query Bi-directional probabilistic roadmap planner with Lazy collision checking (SBL) attempt to detect less explored regions and expand from them [20, 21, 43]. A more recent development is the idea of a Path-Directed Subdivision Tree (PDST) [64]. PDST uses an adaptive subdivision of the state space and a deterministic priority scheme to guarantee coverage, avoiding the use of a metric.

*2) How this expansion should continue*: RRT [42, 60] suggests a Voronoi bias, by expanding toward random states. However, this can become problematic when planning with differential constraints, and controls that achieve specific states cannot be easily computed. Methods that discretize the control set in order to achieve better coverage and reduced planning time have been introduced as well [65,66]. Existence of narrow passages that need to be crossed by valid solutions can significantly reduce the performance of planners. Techniques that improve sampling in narrow passages [67] or identify the direction of narrow passages [68] have also been developed. Recently, the idea of combining two layers of planning has been introduced (SyCLoP, presented as DSLX in earlier work) [61]. SyCLoP is a meta-planner that uses discrete paths (top layer) in a discretization of the workspace to guide the continuous tree exploration (bottom layer). The planner at the bottom layer can be chosen among different tree-based planners.

## 2.2 Task Planning

To solve problems of practical interest it is most often necessary to perform sequences of actions for a robot to achieve its goal. For example, if a humanoid robot is tasked to place a book on a shelve, it will have to perform a minimum of four actions: move one of its arms towards the book, grasp the book, take the book to its destination and release the book. This sequence of operations assumes the robot is capable of three basic actions: move, grasp and release. This assumption is fairly common in previous work and is also made in this thesis. Of course, different sets of actions could have been used to characterize the robot's hardware capabilities. The problem of placing a book on the shelve can be made more interesting if, for example, there is a cup of coffee in the way, such that in order to place the book on the shelve, the coffee needs to be first moved out of the way. In this case, the number of actions the robot has to perform increases. However, it is unclear whether moving the cup of coffee out of the way is always necessary. To represent the possible sequences of actions a robotic device could choose to perform the typical approach is to use graphs. Figure 2.3 shows an example graph, a *task graph* representing the task of retrieving a book, accounting for the possibility of moving the cup of coffee out of the way. The key observation here is that even if a valid sequence

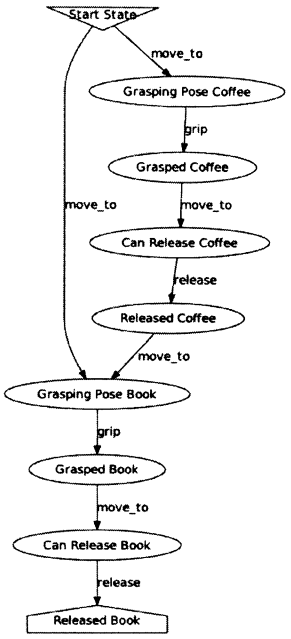



**Figure 2.3:** Example task graph.

of actions is found in the task graph using well known search algorithms [2, 69], that sequence of actions may not be feasible because corresponding motion plans may not exist for that particular sequence of edges (as defined in Section 1.1.2). For this reason, the task and motion planning problems are usually solved simultaneously, as discussed in Section 2.3. In the artificial intelligence domain, task planning implies the construction of the task graphs as well. In this work however, explicit task graphs are assumed as input and the problem addressed is referred to as STAMP.

## 2.3 Task and Motion Planning

There has been a significant amount of research in motion planning (e.g., [3, 4]), grasping (e.g., [70, 71]), task planning (e.g., [2, 69]) and manipulation planning (e.g., [9,72,73]) that has lead to the current state-of-the-art in simultaneous task and motion planning. This section discusses two lines of work:

*1) Work that builds upon the notion of manipulation graph:* The methods included here combine motion and task planning using the notion of a *manipulation graph.* The original description of the manipulation graph [72, 74] is intended for solving the problem of transporting an object of interest to its destination using a robot in a static environment with movable obstacles. Vertices in manipulation graphs correspond to subsets of the state space where the object to be transported is both at a stable configuration (e.g., on a table) and graspable (i.e., the robot can safely hold the object). An edge in the manipulation graph exists if a motion planner can find a plan that connects the edge's vertices. The edges in the manipulation graph

denote one of the following:

- *Transit path*: path executed by the robot without carrying any objects, avoiding collisions with the environment and with itself.
- *Transfer path*: path executed by the robot while carrying an object. The object is considered fixed to the robot along such a path and is treated as part of the robot for collision avoidance purposes.

Solving the task and motion planning problem is then reduced to finding a sequence of transit and transfer paths. A significant amount of previous work expands on the idea of the manipulation graph to produce systems that can solve more complex problems that can include multiple mobile robots (e.g., [9–15, 72–77]).

*2) Work that relies on the idea of guiding continuous exploration with discrete paths:* This line of work performs motion planning in the state space of the robot and uses paths in the task graph as discrete guides (e.g., [78, 79]). Similar concepts have been used for planning foot steps for humanoid robots (e.g., [80]) and climbing robots (e.g., [81]).

The line of work mentioned first is closer to the work in this thesis. Each of the systems that falls under that category employs a representation of a task graph. In this thesis, the distinction between transfer and transit paths is not important. We refer to it only for a more clear connection to previous work.

One of the most complex systems that extends the idea of manipulation graphs is aSyMov [12]. This system is presented in more detail, as it is representative. Planning

can be performed for multiple robots, with potentially different capabilities, under symbolic and geometric constraints in an environment with movable objects. This is a very general form of the problem and allows for solving complex tasks. aSyMov uses a STRIPS-like [2] language to represent tasks. Because probabilistically complete [3] motion planners are used, maximum runtime bounds are imposed in order to make the algorithm terminate. Although performance is good for tasks that involve one robot operating on one object, the aSyMov paper shows the success rate of the method drops to 15% for tasks that involve two robots moving two objects, and computation can take more than a hundred seconds on current modern machines [12]. Even though the latter task is seemingly still simple, the number of sub-tasks and individual motion plans that have to be computed is high, which increases the running time.

The approach aSyMov follows is based on constructing Probabilistic RoadMaps (PRMs) [3, 36] for every robot and every movable object in the environment. A roadmap $R_1$ is connected to a roadmap $R_2$ by identifying a milestone in $R_1$ and creating a corresponding milestone in $R_2$. Use of these connected roadmaps effectively allows the extraction of transit and transfer paths.

aSyMov is not the only system capable of task and motion planning. Over the years, a number of other systems were proposed (e.g., [9–11, 13–15, 72, 73, 75–77]), with varying levels of interaction between task and motion planning. Alami et al. use motion planning as a subroutine and information about its progress is not used at the task planning level [77]. Hauser and Latombe use roadmaps [10] in a similar fashion to Cambon et al. [12]: they are connected by explicitly sampling the intersection of the state spaces they correspond to.

The work by Wolfe et al. [14] uses a hierarchical representation of tasks. At the lowest level of the hierarchy there are primitive actions. Among these actions there can also be algorithms that compute motion plans. This hierarchical representation speeds up the search at the task level. Information about the length of the paths produced by motion planning is used to provide optimal solutions at the task level. A hierarchical representation for performing motion planning with temporal goals is shown by Fainekos et al. [82].

Most of the previous work mentioned so far presents algorithms that can solve the STAMP problem, and many of the approaches also include means of generating task graphs from input specifications. The main difference to the work presented in this thesis is that the option of using subsets of the robot's hardware components to perform actions is not considered. Considering the option of planning for only some of the hardware components of the robot leads to significant computational gains, as shown in Chapter 5.

One of the contributions of Nielsen and Kavraki [73] is that of showing how to compute a sequence of motion plans along a path from an input manipulation graph, in a manner that reduces computation along the path segments that are hard to plan for. This idea is further explored and generalized in this work.

# Chapter 3

# The KPIECE Algorithm for Motion Planning

## 3.1 Space Exploration when Planning under Differential Constraints

When considering dynamic constraints of the robotic system, such as friction, bounds in accelerations, bounded forces, etc., it is typical that motion plans are computed using sampling-based tree planners (e.g., [20, 21, 32, 41–43, 46, 49, 54, 56, 83–85]). These are general algorithms, applicable to a variety of systems, and do not rely on particular properties of robotic systems. Although algorithms capable of solving difficult problems exist, better planning algorithms are needed as robotic systems become more complex and there is a need to account for dynamic constraints. Certain well known techniques for speeding up motion planning algorithms [57] are not

applicable in such cases; for example, consideration of friction requires computation of contacts, which makes lazy collision checking [20, 40] inapplicable; if the model of the robotic system cannot be used for simulation backward in time, the use of bi-directional algorithms is not possible. To quickly compute motion plans for systems with complex dynamics, two approaches can be followed: *(1)* ignore the complexities of the system and only compute geometric paths (sequences of states), in the hope that a controller can follow the paths by keeping velocities sufficiently low (e.g., [86]); *(2)* improve the exploration capabilities of the tree planner through means that only depend on forward propagating the model of motion – numerically evaluating motions only forward in time. While the first approach allows for the implementation of algorithms that can make use of many techniques for speeding up the planning process [57], including bi-directional search and lazy collision checking, the execution of rapid motions or of motions that must account for payload, friction, etc., cannot be correctly planned. In consequence, the latter approach is followed in this chapter.

This chapter presents KPIECE (Kinodynamic Planning by Interior-Exterior Cell Exploration) [87], a tree sampling-based motion planning algorithm, specifically designed for systems with complex dynamics. An initial version of this algorithm has been previously presented by the author [83, 88]. KPIECE was previously shown to be an efficient algorithm in a variety of planning scenarios. The recent developments presented in this chapter make the algorithm more general, applicable for a wider variety of problems and reduce the input required from the user. The presented developments are are also applicable to related algorithms such as EST [21] or SBL [20].

As described in Section 2.1, when searching high-dimensional state spaces using

a tree of motions, deciding which part of the tree merits further exploration can be difficult. Considering dynamics further complicates the problem for a number of reasons: *(1)* the dimensionality of the state space is typically higher because velocities and perhaps accelerations are included in the robot's state, *(2)* the state space may not be entirely reachable from the robot's initial state, *(3)* it is often difficult to define a meaningful metric for these complex state spaces. These problems are addressed by the exploration strategy KPIECE uses, and significant computational advantages are achieved over previous work. The key feature of the exploration strategy used by KPIECE is the estimation of coverage in a Euclidean projection of the state space. This chapter contributes to the further development of KPIECE with a means to automatically compute the projections necessary for the exploration strategy. We now describe how the coverage estimation process works (Section 3.1.1). The KPIECE algorithm is then presented in Section 3.1.2. Section 3.1.3 shows how to automatically compute projections necessary in the exploration process, and experimental results are shown in Section 3.2.

### 3.1.1 Estimating State Space Coverage

The key difficulty in guiding the exploration of the state space when computing motion plans is avoiding over-exploration of certain regions of the state space and under-exploration of other regions of the state space. If the coverage of the state space could be evaluated easily, deciding which parts of the state space should be explored further would be a simpler problem. However, it is not apparent how to quickly evaluate such measures of coverage in general.

To avoid this problem, an approach followed in the literature is to employ a projection of the state space $\mathcal{X}$ to help in making the decision of where to continue the tree expansion from [20,51,56,83]. We denote this projection space by $\mathcal{E}(\mathcal{X})$. Typically $\mathcal{E}(\mathcal{X})$ is assumed to be Euclidean and of low dimension; $\mathcal{E} : \mathcal{X} \rightarrow \mathbb{R}^k$, ($k$ is up to 3 or 4, by today's standards). Estimating the coverage of $\mathcal{E}(\mathcal{X})$ is then an easier problem, since simple approaches based on uniform discretization of $\mathbb{R}^k$ can be used, as shown in Figure 3.1. In the context of tree-based planners, the assumption is that if the tree of motions covers $\mathcal{E}(\mathcal{X})$ well, it also covers $\mathcal{X}$ well. It has not been proven when or whether this assumption holds. However, previous work has shown empirically that the tree exploration can be guided towards the goal region for some user-defined projections [51, 56, 83], and significant computational gains are observed. Nevertheless, using projection spaces to estimate coverage replaces one problem by another: it is not clear how to define the projection $\mathcal{E}$ in general. For this reason, motion planners require the projection $\mathcal{E}$ to be supplied as an input.

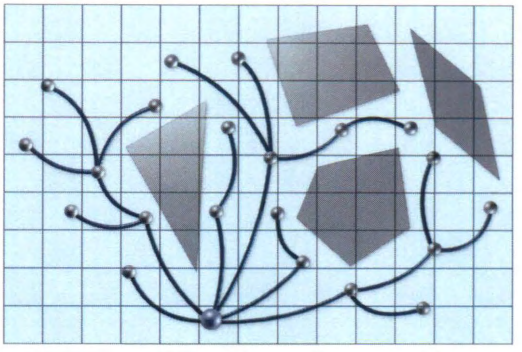


**Figure 3.1:** Uniform discretization of a space to be explored and a possible tree of motions.

Empirical evidence suggests that projections are typically easy to specify because algorithms requiring such projections are usually robust with respect to this input. This statement is supported by experiments conducted with KPIECE, where the same problem instance is solved using different input projections, at similar computational

cost (shown later in Section 3.2).

Intuitively, the purpose of the projection is to provide a space in which coverage is to be estimated, such that the space is representative for the problem being solved. For example, if we are planning for a car moving in plane, a representative space for estimating coverage is the plane in which the car is moving (2-dimensional projection). For a manipulator arm, the position in space of the tip of the arm is representative (3-dimensional projection). For systems where velocity is important, for example an inverted pendulum, a representative space is that of the velocity of the pendulum and its angle (2-dimensional projection). If the pendulum has multiple links, the projection can consist of the norm of angular velocities and the position of the tip of the pendulum in the plane of motion (3-dimensional projection).

### 3.1.2 The KPIECE Algorithm

KPIECE is innovative in the sense that while it is able to handle high-dimensional systems with complex dynamics, it reduces both runtime and memory requirements by making better use of information collected during the planning process. Intuitively, this information is used to decrease the amount of forward propagation the algorithm needs. The key contribution of KPIECE is its exploration strategy. The exploration strategy depends on estimating the coverage of the state space of the robotic system, which in turn depends on the existence of a projection from the state space of the robotic system to a low-dimensional Euclidean space. The initial version of KPIECE assumed such a projection is supplied by the user. The variant presented in this chapter removes this burden from the user and automatically computes such

a projection in a randomized fashion. Before showing how to automatically compute random projections and corresponding experimental results, we describe the basic KPIECE algorithm for convenience. A complete description of the algorithm can be found in [87].

KPIECE is a sampling-based algorithm that explores the state space of the robotic system by growing a tree of motions. Each motion in the tree is defined as $\nu = (s, u, t)$, where $s \in \mathcal{X}$ is the starting state of the motion and $u \in \mathcal{U}$ is the control applied at that state, for a duration $t \in R^{\geq 0}$.

From a high-level perspective, KPIECE proceeds iteratively, as described in Algorithm 1: at each iteration, an existing motion from the tree is selected [line 3]; a new motion starting at a state along the selected motion is produced and added to the tree [line 4]; information gathered in the expansion process is incorporated for future selections of motions [line 7]; this process continues until some termination criterion is met.

The above description is intended to be solely an overview of KPIECE. The steps Algorithm 1 are common to many other sampling-based algorithms that use trees. What makes such algorithms different is how these steps are carried out. In the case of KPIECE, this accounts for up to two orders of magnitude speedup with respect to previous work. Various aspects of KPIECE are discussed in the following paragraphs.

**The Tree of Motions** The tree is initialized with a motion $\nu_0 = (s, \texttt{nil}, 0)$ that consists of the robot's starting state $s \in \mathcal{X}_{valid}$ and a control that has no effect, applied for 0 duration. Although motions are in fact continuous segments, they are computed

**Algorithm 1** KPIECE ($q_{start}$, $N_{iterations}$)

```
1: T = InitializeTree(q_start)
2: for i ← 1...N_iterations do
3:    ν = SelectMotion(T)
4:    ExpandTree(T, ν)
5:    if solution is found then
6:       return solution
7:    EvaluateProgress()
8: return no solution
```

by a forward propagation function (as in Section 1.1.1), with fixed step size. This means that intermediate states along each motion are generated at a fixed resolution. We call this resolution the *propagation step size*. As a result, for every motion, the duration of the control is $t = m \cdot r$, where $r \in \mathbb{R}^+$ is the propagation step size and $m \in \mathbb{N}$ is the number of steps.

The controls applied from $s$ are selected uniformly at random from $\mathcal{U}$. The duration of the control is obtained by sampling a value for $m$. The random selection of controls is what is typically done if other means of control selection are not available. This choice is not part of the proposed algorithm, and can be replaced by other methods, if available. Different methods of control selection are desirable for systems that are not stable for instance, as in this case random selection of controls will likely not lead to valid states. Using some generic forms of control such as LQR is also possible [89].

For a motion $\nu$, let $States(\nu)$ be the set of states along the motion $\nu$, at the propagation step size, as they are generated by forward propagation. $States(\nu)$ is not stored by KPIECE, but it is generated as needed. See Figure 3.2 for an example. New motions expanded from an existing motion $\nu$ can start at any state in $States(\nu)$.

Let $AS = \bigcup_\nu States(\nu)$ be the set of all states that the tree of motions consists of, with respect to the used propagation step size. $AS$ is not computed by KPIECE, but it is a notion used to explain the execution of the algorithm.

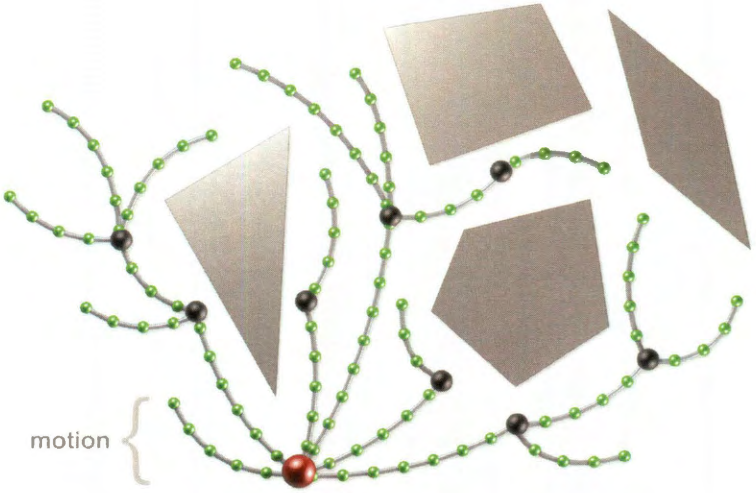



**Figure 3.2:** Tree of motions as grown by KPIECE. The states at the start of motions are depicted as larger vertices. The motion is computed by forward integration at fixed step size. Intermediate states are depicted as smaller vertices. The intermediate states are not stored by KPIECE.

Not unlike other sampling-based planners that employ trees, KPIECE tries to reach the goal as quickly as possible, but also eventually explore entirely the reachable regions of the valid state space $\mathcal{X}_{valid}$, so that a solution is found if one exists. In order to achieve this, KPIECE carefully selects motions for further expansion [line 3 in Algorithm 1]. An important part of the selection strategy is estimating the coverage of the state space that the tree of motions achieves.

**Using Projections to Estimate State Space Coverage** As $\mathcal{X}$ can be high-dimensional and its topology is not known to the algorithm, a projection $\mathcal{E}$ to a Euclidean space $\mathbb{R}^k$ is used. This is an input to the initial version of the algorithm, and Section 3.1.3 will show how to automatically compute this input. $\mathbb{R}^k$ is the projection space and $k$ is the dimension of the projection.

Define $Coord : \mathbb{R}^k \to \mathbb{Z}^k$, where $\mathbb{Z}$ is the set of integers:

$$\begin{aligned} Coord(\mathbf{p}) &= Coord((p_1, \ldots, p_k)) \\ &= \left( \left\lfloor \frac{p_1 - o_1}{d_1} \right\rfloor, \ldots, \left\lfloor \frac{p_k - o_k}{d_k} \right\rfloor \right) = \mathbf{z}, \end{aligned}$$

where $\lfloor \cdot \rfloor$ denotes truncation to nearest smaller integer, $\mathbf{p} = (p_1, \ldots, p_k) \in \mathbb{R}^k$, $\mathbf{o} = (o_1, \ldots, o_k) \in \mathbb{R}^k$ is an arbitrary point designated as the origin, $d_i \in \mathbb{R}^+, i \in \{1, \ldots, k\}$ and $\mathbf{z} \in \mathbb{Z}^k$. $Coord$ discretizes $\mathbb{R}^k$ into $k$-dimensional cubes of uniform size, each with sides of lengths $d_1, \ldots, d_k$.

For every $\mathbf{z} \in \mathbb{Z}^k$, define the corresponding cell in $\mathcal{X}$ to be:

$$Cell(\mathbf{z}) = \{x \in \mathcal{X} \mid Coord(\mathcal{E}(x)) = \mathbf{z}\}.$$

Motions added to the tree of motions are said to be part of a cell if all their states are included in the cell:

$$Motions(\mathbf{z}) = \{\nu \mid x \in States(\nu) \text{ implies } x \in Cell(\mathbf{z})\}.$$

The invariant that every motion is part of a single cell is maintained. This is

achieved by splitting motions before adding them to the tree of motions, such that they are not part of multiple cells. When a motion $\nu$ is to be added, $States(\nu)$ is generated. For every $x \in States(\nu)$, $Coord(\mathcal{E}(x))$ is computed. With this information, it can be decided which parts of the motion go to which cells. Since $States(\nu)$ is an approximation of the motion $\nu$, this computation is not exact, but it is sufficient for our purposes.

It is now possible to define the coverage achieved by a tree of motions in $\mathcal{X}$. For every cell coordinate $\mathbf{z} \in \mathbb{Z}^k$, the coverage of $Cell(\mathbf{z})$ is

$$Coverage(\mathbf{z}) = \sum_{\nu=(s,u,t) \in Motions(\mathbf{z})} \left(1 + \frac{t}{r}\right),$$

where $r$ is the propagation step size. Since $t$ is an integer multiple of $r$, the value of the coverage represents the number of states in $Cell(\mathbf{z})$ that are also in $AS$: $Coverage(\mathbf{z}) = |AS \cap Cell(\mathbf{z})|$, where $|\cdot|$ denotes the cardinality of a set.

At this point we make the assumption that coverage estimates for cells are relevant for the coverage of $\mathcal{X}$. We do not prove this is the case from a mathematical point of view, but experimental results shown later support this hypothesis.

As the tree of motions increases, and the number of states in $AS$ increases, KPIECE keeps track of the minimal set of cells that covers $AS$. We say $C \subset \mathbb{Z}^k$ covers $AS$ if $AS \subseteq \bigcup_{\mathbf{z} \in C} Cell(\mathbf{z})$. We say $C$ is minimal if there is no subset $D \subsetneq C$ such that $D$ covers $AS$. When the algorithm starts, $AS$ has only one state. One cell is sufficient to cover $AS$ – the cell that contains the starting state. Let $M_c \subset \mathbb{Z}^k$ denote the minimal cover of $AS$. Throughout the execution of the algorithm, the

cardinality of $M_c$ increases. Cells included in $M_c$ are called *instantiated cells*. $M_c$ is called a *discretization* of the space covered by the tree of motions.

**Distinguishing Interior and Exterior Cells** For every $\mathbf{z} = (z_1, \ldots, z_k) \in \mathbb{Z}^k$, the neighbors of $Cell(\mathbf{z})$ are $Neighbors(\mathbf{z}) =$

$$\{\, Cell(\mathbf{w}) \in M_c \mid \mathbf{w} = (z_1, \ldots, z_{i-1}, y, z_{i+1}, \ldots z_k), \\ \text{for } y = z_i - 1 \text{ or } y = z_i + 1 \,\}.$$

The maximum cardinality of $Neighbors(\mathbf{z})$ is $2k$. A distinguishing feature of KPIECE is the notion of interior and exterior cells. A cell is considered exterior if it has less than $2k$ neighboring cells. Cells with $2k$ neighboring cells are considered interior. The reason for making this distinction is that focusing the exploration on exterior cells allows the motion planner to cover the state space faster. As the algorithm progresses and new cells are created, some exterior cells become interior (see Figure 3.3). When larger parts of the state space have been explored, most cells have become interior. However, for high-dimensional spaces, to avoid having only exterior cells, the definition of interior cells can be relaxed and cells can be considered interior before the cardinality of the set of neighbors reaches $2k$.

**Importance of Cells** An important step in the execution of KPIECE is selecting the cell from which to continue the expansion of the exploration tree. This section describes the notion of *importance* associated to cells, a notion used in the selection of cells. The following pieces of information are considered when selecting a cell $Cell(\mathbf{z})$:

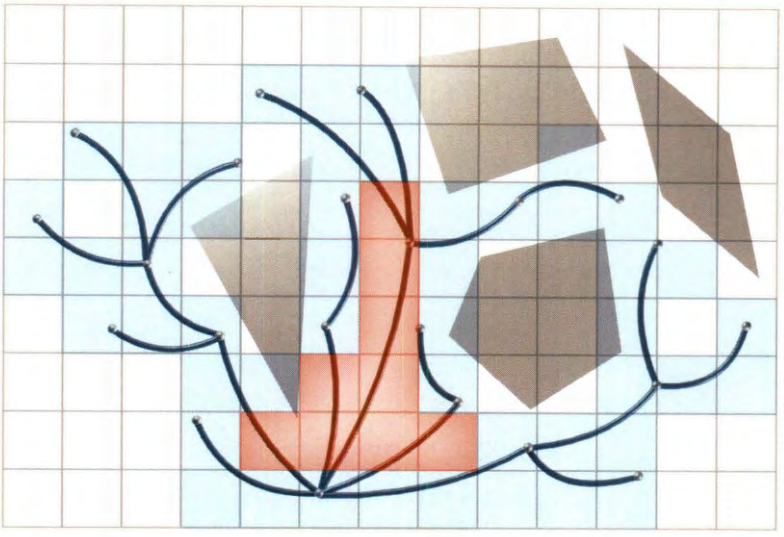


**Figure 3.3:** Representation of a tree of motions and its minimal cover. Interior cells are differentiated from exterior cells.

- The coverage $Coverage(\mathbf{z})$ (work in the same spirit suggested in e.g., [61]),
- The number of times $Cell(\mathbf{z})$ was previously selected (suggested in e.g., [21]),
- The cardinality of $Neighbors(\mathbf{z})$,
- The iteration at which $Cell(\mathbf{z})$ was added to $M_c$ (suggested in e.g., [64]),
- A measure of the progress in exploration achieved when expanding from $Cell(\mathbf{z})$ (work in the same spirit suggested in e.g., [54]).

KPIECE prefers expanding from cells that are less covered rather than from cells that are well covered. Cells that have been selected for expansion fewer times are preferred over cells that have been selected many times. Cells that have fewer neighbors and cells that have been instantiated more recently are preferred, as these are

more likely to be closer to unexplored areas of the space. Cells that have led to good progress in exploration are preferred over cells that have led to slower progress (e.g., if a cell contains many motions that place the robot in front of a wall, it is possible expanded motions will often hit the wall).

The considerations mentioned above for selecting cells are heuristics that have been shown to work well in practice. KPIECE combines their use in the notion of *importance*, since no one heuristic can be identified as better than the others. The importance of a cell $Cell(\mathbf{z})$ is defined as:

$$Importance(\mathbf{z}) = \frac{\log(\mathcal{I}) \cdot \texttt{score}}{\mathcal{S} \cdot (1 + |Neighbors(\mathbf{z})|) \cdot Coverage(\mathbf{z})},$$

where $\mathcal{I}$ is the number of the iteration at which $Cell(\mathbf{z})$ was added to $M_c$, $\mathcal{S}$ is the number of times $Cell(\mathbf{z})$ was selected for expansion (initialized to 1) and `score` reflects the exploration progress achieved when expanding from $Cell(\mathbf{z})$ (initialized to 1). The definition we propose for importance represents what worked well in our experiments. However, it is possible that other definitions can lead to better performance for certain applications. KPIECE prefers expanding from cells with higher importance. With careful bookkeeping, the importance of a cell can be computed in constant time, since all the values it depends on can be made readily available.

To make the definition of importance complete, the use of `score` needs to be explained. Adding a motion to the tree of motions may increase the coverage of the space. The update to `score` proceeds as follows:

- Assume a motion was selected for expansion from $Motions(\mathbf{z})$ (Algorithm 1,

line 3).

- Let total coverage $C_{before} = \sum_{\mathbf{z} \in M_c} Coverage(\mathbf{z})$, and $T_{before} =$ current time.
- Algorithm 1 proceeds with lines 4 through 6.
- Let total coverage $C_{after} = \sum_{\mathbf{z} \in M_c} Coverage(\mathbf{z})$, and $T_{after} =$ current time.
- Line 7 of Algorithm 1 consists of the following steps:

$$\begin{aligned} P &= \alpha + \beta \cdot \left( \frac{C_{after} - C_{before}}{T_{after} - T_{before}} \right) \\ \texttt{score} &= \texttt{score} \cdot \min(P, 1), \end{aligned}$$

  for `score` corresponding to $Cell(\mathbf{z})$.

The purpose of `score` is to reflect how much progress has been made when expanding from $Cell(\mathbf{z})$. Based on the increase in total coverage and the time spent achieving this increase in coverage, a penalization value $P$ is computed. $P$ is used as a multiplicative factor for `score`. To avoid entering an infinite loop where the cell with highest importance is always the same, `score` must never be multiplied by a value larger than 1, hence the use of $\min(P, 1)$. $\alpha$ and $\beta$ are implementation specific constants that help defining $P$. $P$ is intended to be smaller than 1 for expansions that did not provide significant increase in coverage. If $P \geq 1$, the score is not be changed. If the coverage increase is 0, $P = \alpha$, so $\alpha$ must always be larger than 0 so that the score does not become 0. $\beta \in \mathbb{R}^+$ is chosen such that $P$ ends up being larger than 1 only for expansions that have led to significant progress.

**Algorithm Execution** A more detailed description of KPIECE is given in Algorithm 2. The algorithm begins by initializing the tree of motions with a motion of 0 duration that consists solely of the robot's starting state [lines 1–3]. This motion is added to a special data structure called `Grid`. `Grid` associates $Motions(\mathbf{z})$ to every $\mathbf{z} \in M_c$ and takes care of the bookkeeping necessary to update the importance of cells as the algorithm is running.

To expand the tree of motions, KPIECE needs to select an existing motion from that tree. `Grid` is used to identify areas of $\mathcal{X}$ that are considered more important – using the notion of importance defined above [line 5].

KPIECE randomly decides whether to expand from an interior or exterior cell from $M_c$. A strong bias towards exterior cells is usually employed. This decision effectively separates $M_c$ in two disjoint sets: $M_{c,int}$ and $M_{c,ext}$ (the set of interior cells and the set of exterior cells, respectively). Subsequently, KPIECE deterministically selects the cell $Cell(\mathbf{c})$ with maximum importance from either $M_{c,int}$ or $M_{c,ext}$. This operation can be performed quickly by maintaining $M_{c,int}$ and $M_{c,ext}$ as heaps. A motion $\nu$ from $Motions(\mathbf{c})$ is then picked according to a half-normal distribution. The half-normal distribution $h(\sigma^2)$ is used because motions that have been added more recently are preferred for expansion [line 6]. $h(\sigma^2)$ corresponds to the normal distribution (mean = 0 and variance = $\sigma^2$) folded about the $y$-axis at 0; it returns a value larger than 0, with a high probability of being close to 0. For a set $Motions(\mathbf{c})$ with $m$ motions, numbered from 0 to $m-1$, where the $0^{th}$ motion is the most recently added one, a randomly selected motion $\nu$ is the $\lfloor h((m/3)^2) \rfloor^{th}$ motion. A state $s$ along $\nu$ is then chosen uniformly at random from $States(\nu)$ [line 7]. Expanding the tree of

motions continues from $s$ [line 9]. Because $States(\nu)$ is not stored it may be necessary to recompute $s$, but the memory savings outweigh the computational costs.

If the tree expansion was successful, the newly obtained motion is added to the tree of motions and the discretization is updated [lines 11,13]. Information gained during the expansion step is incorporated in the `score` of the selected cell **c**, as previously described [lines 14,15].

**Algorithm 2** KPIECE($q_{start}$, $N_{iterations}$)

1: Let $\nu_0$ be the motion of duration 0 containing solely $q_{start}$
2: Create an empty `Grid` data-structure $G$
3: $G$.AddMotion($\nu_0$)
4: **for** $i \leftarrow 1...N_{iterations}$ **do**
5: $\mathbf{c} = G$.Select(0.75)
6: Select $\nu \in Motions(\mathbf{c})$ using a half-normal distribution
7: Select $s$ along $\nu$
8: Sample random control $u \in \mathcal{U}$ and simulation time $t \in \mathbb{R}^+$
9: Check if any motion $(s, u, t_o)$, $t_o \in (0, t]$ is valid (forward propagation)
10: **if** a motion is found **then**
11: Construct the valid motion $\nu_o = (s, u, t_o)$ with $t_o$ maximal
12: If $\nu_o$ reaches the goal region, **return** path to $\nu_o$
13: $G$.AddMotion($\nu_o$)
14: $P = \alpha + \beta \cdot$ (ratio of increase in coverage to time spent in simulation)
15: Multiply the `score` of $Cell(\mathbf{c})$ by min($P$,1)
16: **return** no solution

### 3.1.3 Using Random Linear Projections

Finding an input projection can be often intuitive, and multiple different projections work well for the same robotic system. For example, for a manipulator arm, in addition to the position in space of the tip of the arm, the projection that only considers the first two angles of the arm (closest to base) also leads to good results.

In some cases however, for example in the case of reconfigurable robots, defining a projection can be hard. In those cases, or when the user is simply unwilling to set a projection, random projections can be automatically computed and used as a fall-back. The inspiration to use random projections comes from a theorem by Johnson and Lindenstrauss [90] which states:

*For any $\varepsilon > 0$, any $n$ points from a $l_2$ metric can be embedded in a $l_2$ metric of dimension $O(\log n/\varepsilon^2)$, with $(1+\varepsilon)$ distortion.*

This means that distances between states in the state space with the $l_2$ norm are approximately preserved in the projection space with the $l_2$ norm. Since $l_2$ norm is usually not an appropriate metric for the state space $\mathcal{X}$, we do not rely on the mathematical foundation provided by this theorem [91].

Automatically computing projections has the potential of finding projections that are better than what even an expert user can provide, in the sense that planners will run faster, it opens up the possibility of using different projections for different environments, and it simplifies the input to motion planning algorithms. Moreover, as the complexity of robotic systems increases, human intuition may fail to produce any useful projections.

**Sampling a Random Linear Projection** We assume a tractable dimension for our projection space; low-dimensional projection spaces are preferred: 2- or 3-dimensional. Using 4- or 5-dimensional projection spaces is possible, but computationally expensive by today's standards. For the purposes of this work, dimensions of $k = 2$ and $k = 3$ were used. Next, $k$ vectors in $\mathbb{R}^n$, where $n > k$ is the dimension of $\mathcal{X}$ (or an

ambient space surrounding $\mathcal{X}$), are randomly sampled according to a normal distribution (with mean 0 and variance 1). Other methods of sampling, such as uniform sampling, would work as well. The $k$ vectors already constitute a projection, but to avoid representing the same information in multiple dimensions, the Gramm-Schmidt process is ran to make the $k$ vectors orthonormal. For a state $x \in \mathcal{X}$, a random linear projection $\mathbf{V}$,

$$\mathbf{V} = (\mathbf{v}_1, ..., \mathbf{v}_k), \ \mathbf{v}_i \in \mathbb{R}^n,$$

the projection of $x$ is $\mathcal{E}(x) \in \mathbb{R}^k$, with

$$\mathcal{E}(x) = \mathbf{V}^T x,$$

assuming all vectors are column vectors. When KPIECE is executed without specifying a projection as input, the process described above can be used to generate a projection. The following section shows the performance of KPIECE with such projections.

## 3.2 Performance of KPIECE with Random Linear Projections

The idea of using random linear projections is simple to apply but the influence of such projections on the performance of KPIECE (and of other algorithms that employ this input) is unclear. To answer this question we propose to simply evaluate

the performance of KPIECE in terms of runtime for a number of sampled random linear projections and for some user specified projections.

### 3.2.1 Experimental Setup

Due to the probabilistic nature of sampling-based algorithms, the runtime of KPIECE needs to be averaged over multiple executions (starting from different random seeds). To reduce the amount of computation necessary for this experiment, a simple method of approximating a projection's utility was defined. Given a trial problem to solve, KPIECE is run twice using a random projection as input. The utility of that projection is then the inverse of the average runtime of the two executions. If at least one of the two executions was unable to reach a solution, the utility is considered to be 0. Sorting sampled projections by utility allows performing a more careful evaluation on only a fraction of the sampled projections, thus saving computational time.

Algorithm 3 shows how to identify four random projections of interest: the three with top utility and the one with median utility. The performance of these four projections is compared with user defined projections on a set of problems. The considered problem instances are shown in Section 3.2.2 and experimental results are in Section 3.2.3.

**Algorithm 3** FindProjection$(k, n)$

**for** $i = 1$ **to** $N_{attempts}$ **do**
  $\mathbf{V}[i] \leftarrow$ RandomLinearProjection$(k, n)$
  $utility[i] \leftarrow$ EvaluateProjection$(\mathbf{V})$
$(R1, R2, R3) \leftarrow$ the 3 projections with highest $utility$
$M \leftarrow$ the projection with median $utility$
**return** $(R1, R2, R3, M)$

### 3.2.2 Robot Models

KPIECE uses physics models for the robots it plans the motion for. For this purpose, the Open Dynamics Engine (ODE) [16] (version 0.10) physics simulation library is used. The used simulation step size was 0.05s. A set of ODE models of increasing complexity are defined in the following section. We consider a robot more complex if it has a higher-dimensional state space.

**Modular Robot** The model characterizes the CKBot modules [92]. Using these modules, different robots can be constructed. Each CKBot module contains one motor. An ODE model for serially linked CKBot modules has been created [32]. The task is to compute the controls for lifting the robot from a vertical down position to a vertical up position for varying number of modules, as shown in Figure 3.4. Each module adds one degree of freedom (DOF). The controls represent torques that are applied by the motors inside the modules. The difficulty of the problem lies in the high dimensionality of the control and state spaces as the number of modules increases, and in the fact that at maximum torque, the motors in the modules are only able to statically lift approximately 5 modules. Consequently, the planner has

to find swinging motions to solve the problem. The state space for a chain modular robot with $m$ modules is $\mathcal{X} = \{\mathbf{x} \mid \mathbf{x} = ((x_1, \dot{x}_1), ..., (x_m, \dot{x}_m))\}$, where $x_i$ is the angle position of module $i$, $i \in \{1, .., m\}$. The employed projection was $\mathcal{E} : \mathcal{X} \rightarrow \mathbb{R}^3$. In the evaluation of $\mathcal{E}$, the first two dimensions are the $(x, z)$ coordinates of the last module ($x, z$ is the plane observed in Figure 3.4) and the third dimension, the square root of the sum of squares of the rotational velocities of all the modules. The environments the system was tested in are shown in Figure 3.4.

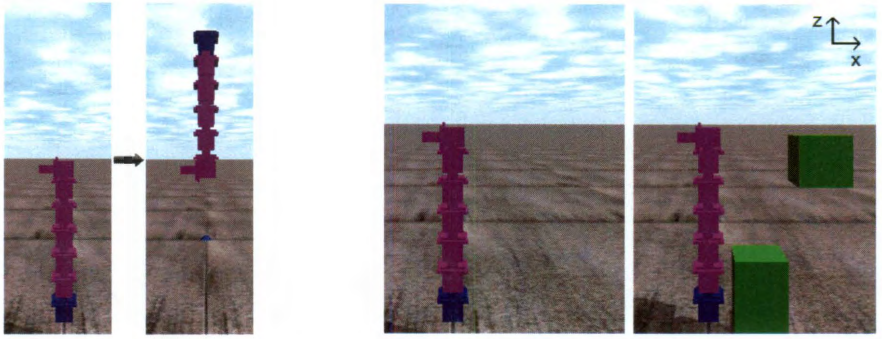



**Figure 3.4:** Left: start and goal configurations. Right: environments used for the chain robot (7 modules). Experiments were conducted for 2 to 10 modules. In the case without obstacles, the environments are named chain1-$x$ where $x$ stands for the number of modules used in the chain. In the case with obstacles, the environments are named chain2-$x$.

**Car Robot** A model of a car [3] was created. The model is fairly simple and consists of five parts: the car body and four wheels. Since ODE does not allow for direct control of accelerations, desired velocities are given as controls for the forward velocity and steering velocity (as recommended by the developers of the library). The desired velocities indicate what velocities the car is intended to achieve and go together with a maximum allowed force. The end result is that the car will

not be able to achieve the desired velocities instantly, due to the limited force. In effect, this makes the system a second order one. The state space for this model is $\mathcal{X} = \{\mathbf{x} \mid \mathbf{x} = (x, y, \theta, v, \dot{\theta})\}$, where $(x, y)$ denote the center of the car chassis, $\theta$ is the car's orientation and $v$ is the velocity along the orientation. The employed projection was $\mathcal{E} : \mathcal{X} \rightarrow \mathbb{R}^2$. $\mathcal{E}$ evaluates to the $(x, y)$ coordinates of the center of the car body. The environments the system was tested in are shown in Figure 3.5.

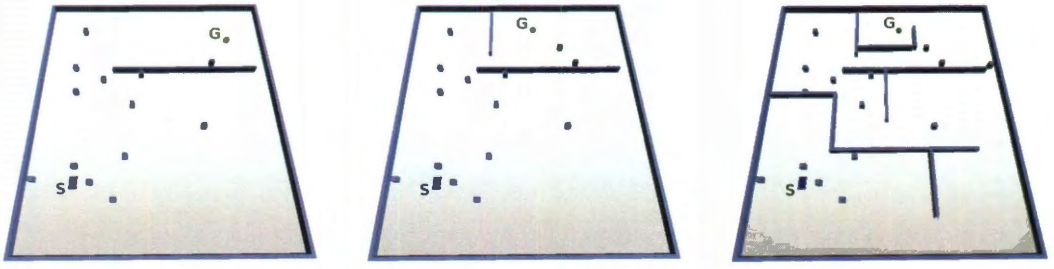



**Figure 3.5:** Environments used for the car robot (car-1, car-2, car-3). Start and goal configurations are marked by "S" and "G". The small cubes represent obstacles.

**Blimp Robot** The third robot that was tested was a blimp robot [62]. The motion in this case is executed in a 3D environment. This robot is particularly constrained in its motion: the blimp must always apply a positive force to move forward (slowing down is caused by drag), it must always apply an upward force to lift itself vertically (descending is caused by gravity) and it can turn left or right with respect to the direction of forward motion. Since ODE does not include air friction, a Stokes model of drag was implemented for the blimp (the drag force is $F_{drag} = -b\mathbf{v}$ where $\mathbf{v}$ is the linear velocity of the blimp and $b$ is the drag coefficient). The state space for this model is $\mathcal{X} = \{\mathbf{x} \mid \mathbf{x} = (x, y, z, \theta, v, \dot{z}, \dot{\theta})\}$, where $(x, y, z)$ denote the center of the

blimp, $\theta$ is the blimp's orientation and $v$ is the forward velocity along the orientation. The employed projection was $\mathcal{E} : \mathcal{X} \rightarrow \mathbb{R}^3$. $\mathcal{E}$ evaluates to the $(x, y, z)$ coordinates of the center of the blimp. The environments the system was tested in are shown in Figure 3.6.

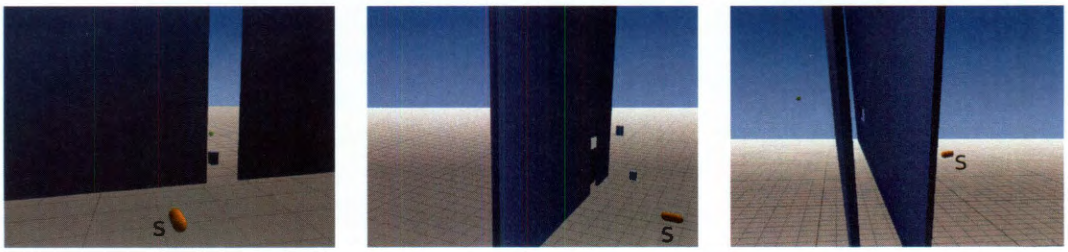



**Figure 3.6:** Environments used for the blimp robot (blimp-1, blimp-2, blimp-3). Start configurations are marked by "S". The blimp has to pass between the walls and through the hole(s), respectively. The small cubes represent obstacles.

### 3.2.3 Results and Discussion

All runtimes reported in this section are the result of running KPIECE with different projections on an 8-core machine, with 16 GB RAM and a 10 minute time limit. For each value, KPIECE was run 50 times; the best 2 and worse 2 results (in terms of runtime) were dropped; the runtime of the remaining 46 runs was averaged to produce the reported value. All values are reported in seconds. The value of $N_{attempts}$ in Algorithm 3 was 150.

Tables 3.1, 3.2 and 3.3 show the averaged runtimes of KPIECE using different projections. The user projections were defined by the author. Every effort was made to find projections that work well. Different combinations of using the velocity of the car and blimp in their projections were attempted, but the best results were obtained

**Table 3.1:** User-defined & 8 random linear projections ($\mathcal{E}$) for the car robot. For each environment, *runtime* (s), *success rate* are reported.

| $\mathcal{E}$ | $k$ | car-1 | car-2 | car-3 |
|---|---|---|---|---|
| U1 | 2 | 7.15, 1.00 | 8.84, 1.00 | 15.90, 1.00 |
| **R1** | **2** | **5.77, 1.00** | **7.93, 1.00** | **14.62, 1.00** |
| $R2$ | 2 | 6.02, 1.00 | 11.13, 1.00 | 37.13, 1.00 |
| $R3$ | 2 | 5.58, 1.00 | 8.82, 1.00 | 24.03, 1.00 |
| $M$ | 2 | 6.30, 1.00 | 8.82, 1.00 | 17.72, 1.00 |
| $R1$ | 3 | 8.04, 1.00 | 10.51, 1.00 | 31.99, 1.00 |
| $R2$ | 3 | 9.27, 1.00 | 12.84, 1.00 | 37.06, 1.00 |
| $R3$ | 3 | 6.22, 1.00 | 7.76, 1.00 | 31.12, 1.00 |
| $M$ | 3 | 6.25, 1.00 | 9.43, 1.00 | 28.82, 1.00 |

**Table 3.2:** User-defined & 8 random linear projections ($\mathcal{E}$) for the blimp robot. For each environment, *runtime* (s), *success rate* are reported.

| $\mathcal{E}$ | $k$ | blimp-1 | blimp-2 | blimp-3 |
|---|---|---|---|---|
| U1 | 3 | 4.04, 1.00 | 7.86, 1.00 | 49.24, 1.00 |
| $R1$ | 2 | 6.69, 1.00 | 132.76, 0.78 | 307.61, 0.13 |
| $R2$ | 2 | 5.42, 1.00 | 15.92, 1.00 | 273.59, 0.43 |
| $R3$ | 2 | 4.12, 1.00 | 12.10, 1.00 | 136.74, 0.59 |
| $M$ | 2 | 10.67, 1.00 | 125.47, 0.93 | 371.79, 0.26 |
| $R1$ | 3 | 3.50, 1.00 | 6.78 , 1.00 | 74.68, 0.98 |
| **R2** | **3** | **3.43, 1.00** | **7.10 , 1.00** | **38.50, 1.00** |
| $R3$ | 3 | 3.52, 1.00 | 30.36, 1.00 | 181.11, 0.65 |
| $M$ | 3 | 3.56, 1.00 | 6.79, 1.00 | 64.55, 1.00 |

with the simplest projections: the workspace. For the modular robot, defining a more complicated projection ($U1$) seems to help more [64]; for comparison purposes, we also define a simple projection ($U2$). The $R1$, $R2$, $R3$ and $M$ projections were obtained as discussed earlier, by projecting from an ambient space surrounding $\mathcal{X}$ to 2- and 3-dimensional projection spaces ($k = 2$, $k = 3$).

Random linear projection that performed best are marked in bold-face. We ob-

**Table 3.3:** User-defined & 8 random linear projections ($\mathcal{E}$) for each modular robot. For each environment, *runtime* (s), *success rate* are reported.

| $N$ | $\mathcal{E}$ | $k$ | chain1-$N$ | chain2-$N$ | $N$ | chain1-$N$ | chain2-$N$ |
|---|---|---|---|---|---|---|---|
| 5 | U1 | 3 | 3.11, 1.00 | 3.14, 1.00 | 8 | 6.04, 1.00 | 30.35, 1.00 |
| | U2 | 2 | 3.24, 1.00 | 20.71, 0.76 | | N/A, 0.00 | N/A, 0.00 |
| | $R1$ | 2 | 3.63, 1.00 | 29.67, 0.87 | | N/A, 0.00 | N/A, 0.00 |
| | $R2$ | 2 | 3.72, 1.00 | 51.22, 0.46 | | 31.14, 0.17 | N/A, 0.00 |
| | $R3$ | 2 | 3.33, 1.00 | 24.66, 1.00 | | 62.67, 0.13 | N/A, 0.00 |
| | $M$ | 2 | 3.64, 1.00 | 86.40, 0.61 | | 155.81, 0.28 | N/A, 0.00 |
| | $R1$ | 3 | 5.80, 1.00 | 26.68, 1.00 | | 197.12, 0.09 | N/A, 0.00 |
| | **R2** | **3** | **5.10, 1.00** | **9.20, 1.00** | | **39.49, 1.00** | **52.70, 0.96** |
| | $R3$ | 3 | 5.90, 1.00 | 21.68, 1.00 | | 162.81, 0.76 | N/A, 0.00 |
| | $M$ | 3 | 6.63, 1.00 | 17.40, 1.00 | | 212.03, 0.26 | 182.23, 0.26 |
| 6 | U1 | 3 | 3.26, 1.00 | 3.34, 1.00 | 9 | 37.24, 1.00 | 133.84, 0.48 |
| | U2 | 2 | 24.35, 0.96 | 85.41, 0.33 | | N/A, 0.00 | N/A, 0.00 |
| | $R1$ | 2 | 150.18, 0.78 | 41.48, 0.13 | | N/A, 0.00 | N/A, 0.00 |
| | $R2$ | 2 | 108.01, 0.65 | N/A, 0.00 | | N/A, 0.00 | N/A, 0.00 |
| | $R3$ | 2 | 3.79, 1.00 | 19.42, 1.00 | | N/A, 0.00 | N/A, 0.00 |
| | $M$ | 2 | 25.18, 1.00 | 109.74, 0.65 | | N/A, 0.00 | N/A, 0.00 |
| | **R1** | **3** | **7.94, 1.00** | **7.68, 1.00** | | **185.90, 0.67** | **144.51, 0.93** |
| | $R2$ | 3 | 8.48, 1.00 | 8.59, 1.00 | | 139.60, 0.41 | 228.04, 0.13 |
| | $R3$ | 3 | 10.10, 1.00 | 60.09, 1.00 | | 201.33, 0.43 | 257.78, 0.76 |
| | $M$ | 3 | 38.65, 1.00 | 131.63, 0.39 | | N/A, 0.0 | N/A, 0.0 |
| 7 | U1 | 3 | 3.92, 1.00 | 4.55, 1.00 | 10 | 214.85, 0.41 | 656.44, 0.02 |
| | U2 | 2 | 120.40, 0.04 | N/A, 0.00 | | N/A, 0.00 | N/A, 0.00 |
| | $R1$ | 2 | 10.06, 1.00 | 74.95, 0.67 | | N/A, 0.00 | N/A, 0.00 |
| | $R2$ | 2 | 108.20, 0.20 | N/A, 0.00 | | N/A, 0.00 | N/A, 0.00 |
| | $R3$ | 2 | 15.33, 1.00 | 146.62, 0.35 | | N/A, 0.00 | N/A, 0.00 |
| | $M$ | 2 | 74.60, 0.74 | 409.50, 0.02 | | N/A, 0.00 | N/A, 0.00 |
| | $R1$ | 3 | 19.88, 1.00 | 30.32, 1.00 | | N/A, 0.00 | N/A, 0.00 |
| | **R2** | **3** | **18.62, 1.00** | **23.27, 1.00** | | N/A, 0.00 | N/A, 0.00 |
| | $R3$ | 3 | 34.67, 1.00 | 41.67, 1.00 | | N/A, 0.00 | N/A, 0.00 |
| | $M$ | 3 | 72.12, 1.00 | 138.81, 0.11 | | N/A, 0.00 | N/A, 0.00 |

**Table 3.4:** The percentage of the projections that were considered valid by the evaluation procedure in Section 3.2.3. Maximum allowed time per trial environment is presented as well.

| | 2 dimensions | | 3 dimensions | |
|---|---|---|---|---|
| Trial environment | valid | time (s) | valid | time (s) |
| car-3 | 52.7% | 90.0 | 78.0% | 90.0 |
| blimp-3 | 83.3% | 90.0 | 64.7% | 90.0 |
| chain1-5 | 100.0% | 90.0 | 100.0% | 90.0 |
| chain1-6 | 84.0% | 90.0 | 86.7% | 90.0 |
| chain1-7 | 42.7% | 90.0 | 47.3% | 90.0 |
| chain1-8 | 10.0% | 90.0 | 15.3% | 90.0 |
| chain1-9 | 0.7% | 200.0 | 3.3% | 200.0 |
| chain1-10 | 0.0% | 600.0 | 0.0% | 600.0 |

serve that in the case of the car and the blimp, the random projections actually do a little better than the user-defined projections. This in itself represents an impressive result, considering the simplicity of the process through which the random projections were found. In addition, looking at Table 3.4, we notice that the percentage of random linear projections that produce some results, is very high (above 50%). This means that for systems of moderate dimension, finding a good random linear projection is an easy task. Further evidence supporting this observation is the fact that the $M$ projections also perform well. Of course, there may be other potentially non-linear projections that could do better.

Looking at Table 3.3, where we test systems with higher-dimensional state spaces, random linear projections do not perform as well as the non-linear user-defined projection $U1$. However, it should be noted that finding $U1$ required significant effort. For 5, 6 and 7 modules we do however get results that are no worse than 5 times

slower, with 100% success rate. For 8 modules, we get similar results in terms of runtime but the success rate drops under 100%. At 9 modules an interesting result is observed. With the hard to find $U1$ projection, the success rate is 0.48 (48%) for the chain2-9 environment while with the best found random linear projection, the average runtime is almost the same but the success rate is much higher: 0.93 (93%). At 10 modules, no randomly sampled projections had utility above 0, even though we increased the allowed runtime for the trial environment (as shown in Table 3.4). However, even with the $U1$ projection, we obtain poor results (low success rate). The fact that the runtime is limited to 10 minutes is likely the primary cause for this low success rate. It is possible that using a higher-dimensional projection would also improve results. Comparing with the user-defined projection $U2$, random linear projections do significantly better.

Overall, for systems with moderate dimension it is likely that easy to find random linear projections will perform well. As the dimension increases, this is no longer the case, but we still get reasonable results.

# Chapter 4

# The Open Motion Planning Library and Practical Applications

Many of the core concepts in motion planning are relatively easy to explain, but implementing motion planning algorithms in a generic way is non-trivial. This chapter describes the Open Motion Planning Library (OMPL), an open source C++ implementation (with Python bindings) of many sampling-based algorithms, including KPIECE, and core low-level data structures that are commonly used [93]. OMPL is designed to be used in both academic and industrial settings.

Within the robotics community, it is often challenging to demonstrate that a new motion planning algorithm is an improvement over existing methods according to some metric. First, it is a substantial amount of work for a researcher to implement not only the new algorithm, but also one or more state-of-the-art motion planning algorithms to compare against. Ideally, implementations of low-level data structures

and subroutines used by these algorithms (e.g., proximity data structures) are shared, so that only differences of the high-level algorithm are measured. Second, for an accurate comparison, one needs a known set of benchmark problems. Finally, collecting various performance metrics for several planners with different parameter settings, running on several benchmark problems and storing them in a way that facilitates easy analysis subsequently is a non-trivial task. OMPL was designed to help with all these issues, and make it easier to try out new ideas.

From the beginning, OMPL was intended to be useful in practical applications. This requires that planning algorithms have to be able to solve motion planning problems for systems with many degrees of freedom at interactive speeds. An additional requirement is the ability to cleanly integrate OMPL with other software components on a robot, such as perception, kinematics, control, etc. Through a collaboration with Willow Garage, OMPL is integrated with ROS [22] and serves as the motion planning back-end for the arm planning software stack.

**Related Software Packages for Motion Planning** Several other packages for motion planning are available. Some, such as the Motion Strategies Library (MSL) [94], the Motion Planning Kit (MPK) [95], and VIZMO++ [96], are no longer maintained. Others, such as KineoWorks [97] and the Object-Oriented Programming System for Motion Planning (OOPSMP) [98], are designed as standalone applications for motion planning, which makes their integration with other software components more difficult. OOPSMP is in some sense the predecessor of OMPL, as it significantly influenced the design of OMPL.

Another software package that is related to OMPL is OpenRAVE [99]. OpenRAVE is open source, actively developed, and it is widely used. It is important to understand the difference in design philosophy behind OMPL and OpenRAVE. OpenRAVE is designed to be a complete package for robotics. It includes, among other things: motion planning algorithms, geometry representation, collision checking, grasp planning, forward and inverse kinematics for several robots, controllers, simulated sensors, visualization tools, etc. OMPL, on the other hand, was designed to focus completely on motion planning with a clear mapping between theoretical concepts in the literature and abstract classes in the implementation. This high level of abstraction makes it easy to integrate OMPL with a variety of front-ends and other libraries. Some integration examples are described in Section 4.4. To some extent, the integration with ROS [22] gives a user many of OpenRAVE's features that are purposefully not included in OMPL. As a result of this narrower focus in OMPL, more resources have been spent on implementing a broader variety of sampling-based algorithms than what is currently available in OpenRAVE.

**Conceptual Overview of OMPL** OMPL is intended for use in research and education, as well as in industry. For this reason, the main design criteria for OMPL are as follows:

- **Clarity of concepts** OMPL was designed to consist of a set of components as indicated in Figure 4.1, such that each component corresponds to known concepts in sampling-based motion planning [3].
- **Efficiency** OMPL has been implemented entirely in C++ and is thread-safe.

- **Simple integration with other software packages** To facilitate the integration with other software libraries, OMPL offers abstract interfaces that can be implemented by the "host" software package. Furthermore, the dependencies of OMPL are minimal: only the Boost libraries [100] are required.

- **Straightforward integration of external contributions** API constraints for planning algorithms are minimal, so that external contributions can be easily integrated.

As opposed to all other existing motion planning software libraries, OMPL does not include a representation of workspaces or of robots; as a result, it also does not include a collision checker or any means of visualization. OMPL is reduced to only motion planning algorithms. The advantage of this minimalist approach is that it allowed designing a library that can be used for generic search in high-dimensional continuous spaces subject to complex constraints. Instead of defining valid states as collision-free, which would require a specific geometric representation of the environment and robot as well as support for a specific collision checker, OMPL leaves the definition of state validity completely up to the user. This gives enormous design freedom: the user can defer collision checking to a physics engine, write a state sampler that constructs only valid states, or define state validity in completely arbitrary ways that may or may not depend on geometry.

To make OMPL as easy to use as possible, various parameters needed for tuning sampling-based motion planners are automatically computed. The user has the option to override defaults, but that is not a requirement.

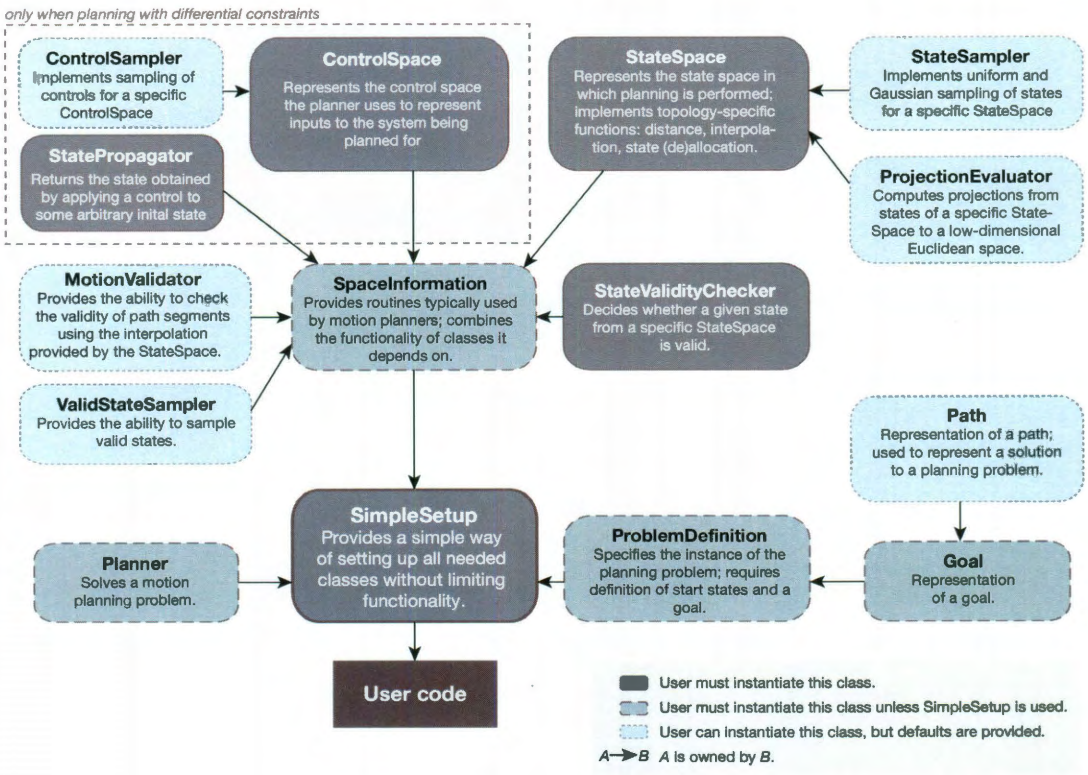



**Figure 4.1:** Overview of OMPL structure

## 4.1 Implementation of Core Concepts

This section describes some of the more important concepts present in OMPL. Figure 4.1 gives a high-level overview of the main classes in OMPL and of their relationships. Class names are written in a sans-serif font (e.g., StateSpace), while methods and functions names are written in a monospaced font (e.g., `isSatisfied()`). For conciseness, the arguments to methods and functions are omitted.

**States, Controls, and Spaces** To maximize the range of application for the included planning algorithms, OMPL represents the search spaces, i.e, the state spaces (denoted as $\mathcal{X}$ earlier) to be searched (StateSpace), in a generic way. State spaces include operations on states such as distance evaluation, test for equality, interpolation, as well as memory management for states: (de)allocation and copying. Additionally, each state space has its own storage format for states, which is not exposed outside the implementation of the state space itself. To operate on states, the planning algorithms implemented in OMPL rely only on the generic functionality offered by state spaces. This approach enables planning algorithms in OMPL to be applicable to any state spaces that may be defined, as long as the expected generic functionality is provided.

Furthermore, OMPL includes a means of combining state spaces using the class CompoundStateSpace. A combined state space implements the functionality of a regular state space on top of the corresponding functionality from the maintained set of state spaces. This allows trivial construction of more complex state spaces from simpler ones. For example SE3StateSpace is just a combination of SO3StateSpace and RealVectorStateSpace. Instances of CompoundStateSpace can be constructed at run time, which is necessary for constructing a state space from an input file specification, as is done, for example, in ROS.

In addition to states and state spaces, some algorithms in OMPL require a means to represent controls. Control spaces (ControlSpace, denoted as $\mathcal{U}$ earlier) mirror the structure of state spaces and provide functionality specific to controls, so that planning algorithms can be implemented in a generic way.

State spaces optionally include specifications of projections to Euclidean spaces (ProjectionEvaluator). Low-dimensional Euclidean projections are used by several sampling-based planning algorithms (e.g., KPIECE, SBL, EST) to guide their search for a feasible path, as it is much easier to keep track of coverage (i.e., which areas have been sufficiently explored and which areas should be explored further) in such low-dimensional spaces.

**State Validation and Propagation** Whether a state is valid or not depends on the planning context. In many cases state validity simply means that a robot is not in collision with any obstacles, but in general any condition on a state can be used. Testing whether a state is valid is achieved through the StateValidityChecker abstract class. In OMPL.app (see Section 4.4) a state validity checker for rigid body motion planning is predefined, but in general, a user needs to implement their own. In addition to testing the validity of states, motion segments (between two states) need to be tested as well. This second validity test is achieved through the MotionValidator class. Based on a given state validity checker, a default MotionValidator is constructed, one that checks whether the interpolation between two states at a certain resolution produces states that are all valid. However, it is possible to plug in a different MotionValidator. For example, one might want to add support for continuous collision checking, which can adaptively check for collisions and provide *exact* guarantees for state validity [101].

For planning with controls, a user needs to specify how the system evolves when certain controls are applied for some period of time starting from a given state. This

functionality is implemented through the StatePropagator class. In the simplest case, a state propagator is essentially a lightweight wrapper around a numerical integrator for systems of the form $\dot{q} = f(q, u)$, where $q$ is a state vector and $u$ a vector of controls. One can use, e.g., standard numerical integrators such as those available from the GNU Scientific Library, variational integrators [102], or a physics engine to perform state propagation.

**Samplers** The fundamental operation that sampling-based planners perform is sampling the space that is explored. Additionally, when considering controls in the planning process, sampling controls may be performed as well.

To support sampling functionality, OMPL includes three types of samplers: state space samplers (StateSampler), valid state samplers (ValidStateSampler) and control samplers (ControlSampler).

State space samplers are implemented as part of the StateSpace they can sample, since they need to be aware of the structure of the states in that space. For instance, uniformly sampling 3D orientations is dependent on their parametrization. Three sampling distributions are implemented by every state space sampler: uniform, Gaussian and uniform in the vicinity of a specified point. This first sampler is necessary to sample over the entire space, while the latter two are used for sampling states near a previously generated state. This is the most basic level of sampling.

Previous work has shown that the strategy used for sampling valid states in the state space significantly influences the runtime of many planning algorithms. Valid state samplers provide the interface for implementing different sampling strategies.

The probability distribution of these samplers depends on the algorithm used and is not imposed as part of the API. The implementation of valid state samplers relies on the existence of a state space sampler and a state validator (StateValidityChecker). A common approach to constructing valid state samplers is to repeatedly call a state space sampler until the state validator returns true. In OMPL there are several valid state samplers implemented: a uniform valid state sampler (UniformValidStateSampler), two samplers (GaussianValidStateSampler, ObstacleBasedValidStateSampler) that generate valid samples near invalid ones (which is often helpful in finding paths through narrow passages [103, 104]).

When considering controls in the planning process, a means to generate controls is also necessary. This functionality is attained using control samplers, which are implemented as part of the control spaces (ControlSpace) they represent.

**Goal Representations** OMPL uses a hierarchical representation of goals. In the most general case, a Goal can be defined by an `isSatisfied()` function that when given a state, reports whether that state is a goal state or not. While this very general implicit representation is possible, it offers planners no indication of how to reach the goal region. For this reason, `isSatisfied()` optionally reports a heuristic distance to the goal region, which is not required to be a metric.

GoalRegion is a refinement of the general Goal representation, one that explicitly specifies the distance to the goal using a `distanceGoal()` function. The `isSatisfied()` function is then defined to return true when `distanceGoal()` reports distances smaller than an user set threshold. GoalRegion is still a very general representation but allows

planners to bias their search towards the goal. A refinement of GoalRegion is GoalSampleableRegion, one which additionally allows drawing samples from the goal region. GoalState and GoalStates are concrete implementations of GoalSampleableRegion.

For practical applications it is often possible to sample the goal region, but the sampling process may be relatively slow (e.g., when using numerical inverse kinematics solvers). For this reason a further refinement of GoalStates is defined: GoalLazySamples. This refinement continuously draws samples in a separate sampling thread, and allows planners to draw samples from the goal region without waiting, after at least one sample has been produced by the sampling thread.

**Planning Algorithms** OMPL includes two types of motion planners: ones that do not consider controls when planning and ones that do. We chose to split the planning algorithms in OMPL in these two categories for efficiency reasons. With additional levels of abstraction it would have been possible to avoid this split [98]. The downside would have been that the implementation of planners would have had to follow a strict structure, which makes the implementation of new algorithms more difficult and possibly less efficient.

If controls are not considered, the solution path is constructed from a finite set of segments, and each segment is computed by interpolation between a pair of sampled states. This type of planners is typically used for computing motion plans under geometric constraints solely. Several geometric planning algorithms are implemented in OMPL, including KPIECE [83, 87], bidirectional KPIECE, bidirectional lazy KPIECE, RRT [60], RRT-Connect [41], lazy RRT, SBL [20], EST [21], and a

```
space = SE3StateSpace()
# set the bounds (code omitted)

ss = SimpleSetup(space)
# "isStateValid" is a user-supplied function
ss.setStateValidityChecker(isStateValid)

start = State(space)
goal = State(space)
# set the start & goal states to some values
# (code omitted)

ss.setStartAndGoalStates(start, goal)
solved = ss.solve(1.0)
if solved:
    print setup.getSolutionPath()
```

```
StateSpacePtr space(new SE3StateSpace());
// set the bounds (code omitted)

SimpleSetup ss(space);
// "isStateValid" is a user-supplied function
ss.setStateValidityChecker(isStateValid);

ScopedState<SE3StateSpace> start(space);
ScopedState<SE3StateSpace> goal(space);
// set the start & goal states to some values
// (code omitted)

ss.setStartAndGoalStates(start, goal);
bool solved = ss.solve(1.0);
if (solved)
    setup.getSolutionPath().print(std::cout);
```

**Figure 4.2:** Solving a motion planning problem with OMPL in Python (left) and in C++ (right).

basic version of PRM [36]. In addition, there are multi-threaded versions of RRT and SBL.

When controls are considered, the solution path is constructed from a sequence of controls. Control-based planners are typically used when motion plans need to respect differential constraints as well. Several algorithms for planning with differential constraints are implemented in OMPL as well, including KPIECE [83,87] and RRT [42].

## 4.2 Example Usage

Figure 4.2 shows the complete code necessary for planning the motion of a rigid body between two states, in Python and C++. In both cases, the only steps taken in the code are: instantiate the space to plan in (SE(3)), create a simple planning context (using SimpleSetup), specify a Boolean operator that distinguishes valid states, specify

the input start and goal states, and finally, compute the solution. The SimpleSetup class initializes instantiations of the core motion planning classes shown in Figure 4.1 with reasonable defaults, which can be overridden by the user if desired.

Essentially, the execution of the code can be reduced to three simple steps: *(1)* specify the space in which planning is to be performed, *(2)* specify what constitutes a valid state, and *(3)* specify the input start and goal states. Such simple specifications are desirable for many users who simply wish motion planning to work, without having to select problem specific parameters, or different sampling strategies, different planners, etc. This capability is made possible by OMPL's automatic computation of planning parameters. In the example above, a planner is automatically selected based on the specification of the goal and the space to plan in. The selected planner is then automatically configured by computing reasonable default settings that depend on the planning context. If a user decides to choose their own planner, or set their own parameters, OMPL allows the user to do so completely—no parameter is hidden.

## 4.3 Benchmarking with OMPL

A seemingly simple but often ignored part of motion planning software is benchmarking planning code. OMPL includes benchmarking capabilities (through a class called Benchmark) that can be simply dropped in and applied to existing planning contexts. In very simple terms, a Benchmark object runs a number of planners multiple times on a user specified planning context represented with SimpleSetup. Although simple, this code automatically keeps track of all the used settings and takes all the

possible measurements during planning (currently, tens of parameters are recorded for every single motion plan). The recorded information is logged and can be post-processed using a Python script included with OMPL. The script can produce MySQL databases with all experiment data so that the user can write their own queries later on, but it can also automatically generate simple box plots for real- and integer-valued measurements, as shown in Figure 4.3, and bar plots for binary-valued measurements.

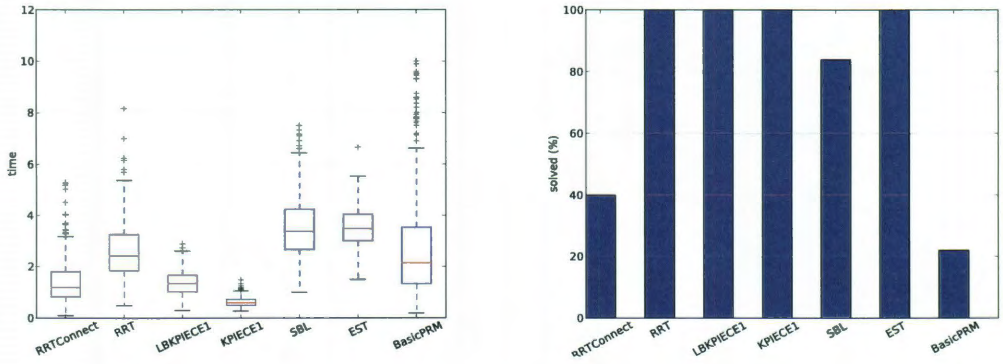



**Figure 4.3:** Sample output of automatically computed benchmark results.

## 4.4 Integration with Other Robotics Software

It is relatively straightforward to integrate OMPL with other robotics software. Below we present two case studies that highlight different use cases.

**OMPL.app: A Graphical User Interface for OMPL** A graphical front-end for OMPL called OMPL.app was created. OMPL.app serves two purposes: *(1)* provide novice users with an easy-to-use interface so they can experiment with several motion

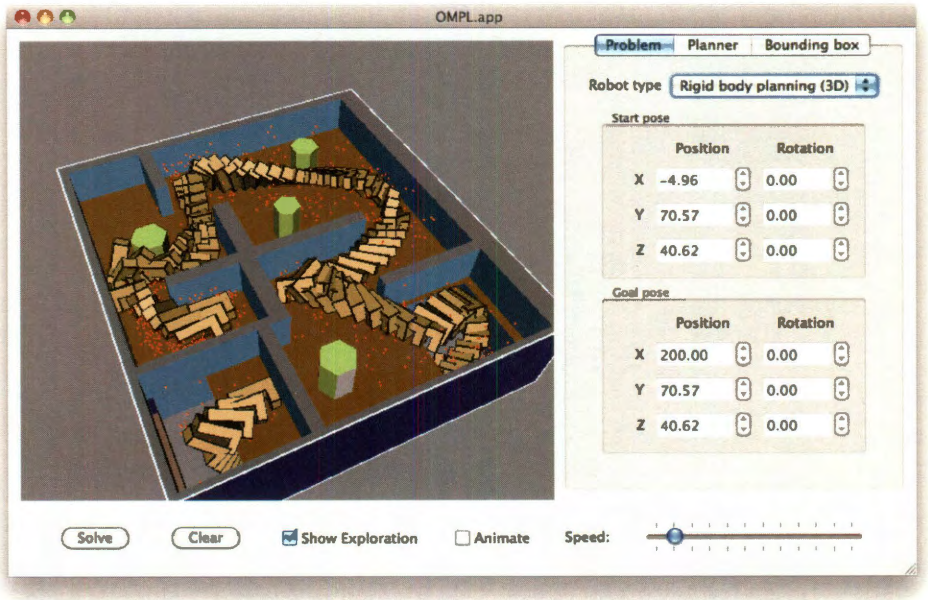



**Figure 4.4:** The OMPL.app graphical interface. A solution path for an L-shaped, free-flying robot is shown. The red dots indicate the positions of sampled states. A user can load meshes that represent the environment and a robot, define start and goal states and solve problems.

planning algorithms and apply them to example rigid body motion planning problems, and *(2)* demonstrate the integration of OMPL with third-party libraries for collision checking and visualization tools. The graphical interface of OMPL.app is shown in Figure 4.4.

**Integration with ROS** OMPL is provided as a ROS package as well, and it is included in the arm planning software stack in ROS. OMPL is interfaced with collision checking, visualization and control components included in ROS. Given a robot de-

scription in URDF[1] format, a state space representation is automatically constructed for OMPL and motion plans can be computed for any user-specified group of joints. Typically, the chosen groups of joints are the seven joints of the arms, although motion planning for the mobile base was also tried successfully.

## 4.5 Applications of OMPL

### 4.5.1 KPIECE for Planning under Geometric Constraints

KPIECE was designed to be used for motion planning with differential constraints. However, this does not mean the algorithm cannot be used for planning under geometric constraints as well. Furthermore, comparisons with certain types of algorithms, such as ones that use lazy collision checking or bi-directional search, cannot be performed for the problems with differential constraints presented above. For KPIECE to be used for planning under geometric constraints, two changes need to be made: *(1)* the sampling of controls to be applied to states $x$ in the tree is replaced by the sampling of random states $x'$, such that $x'$ is nearby $x$ (e.g., a Gaussian distribution that has $x$ as mean and the variance specified as user input can be used to sample $x'$), and *(2)* the simulation of a robot model forward in time under specified controls is replaced by a local planner.

To shed some light on the performance of KPIECE when planning solely under geometric constraints, two experiments with only kinematic constraints are included. The first experiment is that of moving an arm with 7 degrees of freedom (The PR2

[1] `http://www.ros.org/wiki/urdf`

arm from Willow Garage) from a position above a table to a position under the table, as show in Figure 4.5. OMPL integrated with ROS was used for this experiment. The second experiment is that of moving a rigid body from a start configuration to a goal configuration in a complex environment, as shown in Figure 4.6. OMPL.app was used for this experiment. The projection used in the first experiment was a two-dimensional one, consisting of the joint values at the first two joints of the arm. For the second experiment, the projection was the position of the rigid body in space (ignoring orientation).

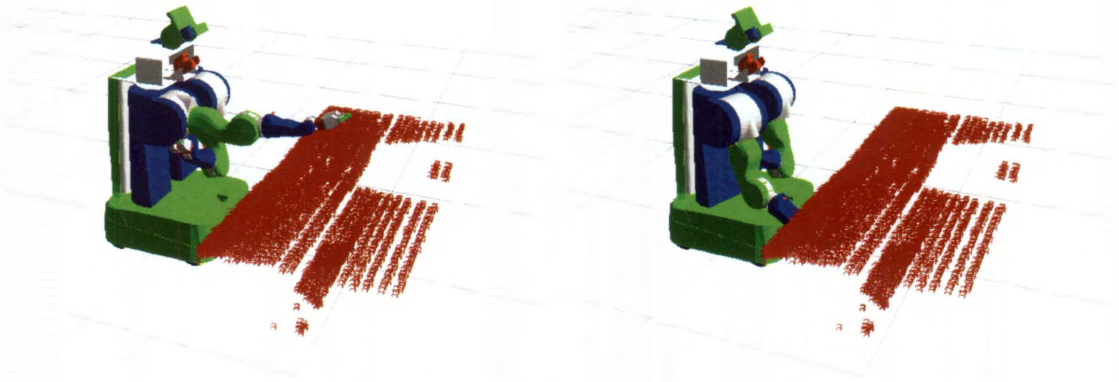


**Figure 4.5:** Move the right arm from above to below the table: start state (left) and goal state (right). The representation of the table is as observed using a laser scanner.

Table 4.1 shows the runtimes of various algorithms when planning for the problems described above, averaged over 100 runs. KPIECE is still faster than RRT and EST, but the speedup is not as significant as in the previous examples. For comparison, runtimes of bi-directional search algorithms, SBL [20] and RRTConnect [41], are included. LBKPIECE is a lazy bi-directional implementation of KPIECE, with a

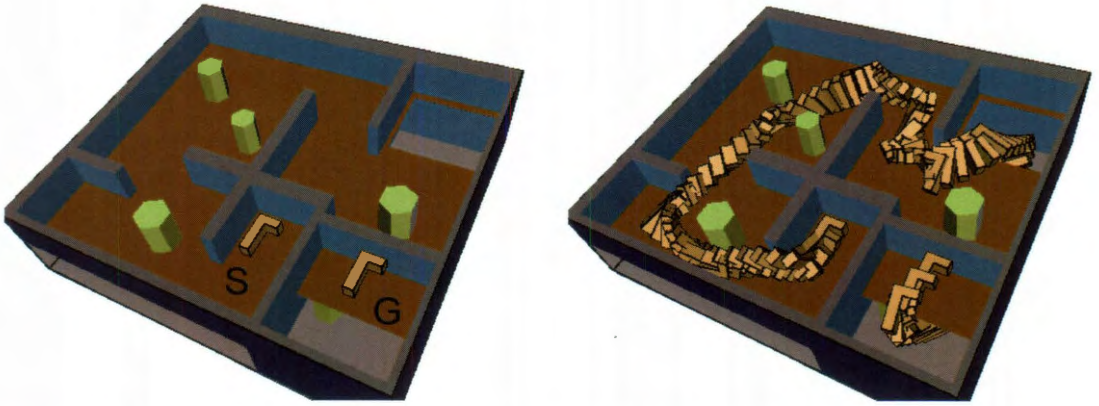



**Figure 4.6:** Move the "L"-shaped rigid body from start to goal, indicated by "S" and "G", respectively.

**Table 4.1:** Runtimes of kinematic versions of the algorithms.

| Algorithm | Arm Plan Time (ms) | Rigid Body Plan Time (ms) |
|---|---|---|
| RRT | 456 | 3248 |
| EST | 187 | 3907 |
| KPIECE | 166 | 698 |
| RRTConnect | 21 | 1508 |
| SBL | 29 | 3943 |
| LBKPIECE | 37 | 1146 |

connection strategy similar to that of SBL. For the arm problem, the bi-directional versions are an order of magnitude faster, with RRTConnect outperforming the other algorithms. For the rigid body problem, since the start and goal states are close in the workspace, bi-directional algorithms no longer perform as well. In fact KPIECE performs best due to its ability to expand towards unexplored space.

### 4.5.2 Integrating Motion Planning with Perception and Control

A key factor to be considered when computing motion plans for real hardware platforms is whether the environment is fixed or not.

If the environment is fixed, then motion planning can be performed offline and there is no need for perception, other than perhaps motor encoders to be used in feedback control. Such approaches are for example applicable for industrial robots, where the same motion needs to be executed repeatedly. Consideration of robot dynamics is necessary depending on the capabilities of the controller and the speed at which the robot needs to be operated. In either case, since the motion planning step can be performed offline, the constraints on the runtime of the motion planner are not too stringent.

If the environment is changing, typically the robot also needs to react to changes in the environment, as they are perceived. In this case, sensors are used to construct representations of the environment around the robot. The representation of the environment is updated often and thus motion plans may need to be updated or recomputed. Furthermore, the perceived robot data is typically in the form of a set of 3D points (a *point cloud*) that is not exhaustive (i.e., does not represent the environment fully) and includes noise. This presents an additional set of difficulties in terms of modeling this data for motion planning.

This section includes experimental validation of the KPIECE algorithm in a real life scenario that accounts for fast-changing environments. To test KPIECE and a bi-

directional version of KPIECE (LBKPIECE) in such scenarios, OMPL was integrated with a perception pipeline on the PR2 from Willow Garage [33, 105]. The PR2 comprises an omni-directional wheeled base, telescoping spine, two force-controlled 7-DOF arms and an actuated sensor head. Each arm has a 1-DOF gripper attached to it. The robot can negotiate ADA-compliant[2] wheelchair-accessible environments, and its manipulation workspace is similar to that of an average-height adult. The sensor head comprises a Hokuyo UTM-30 planar laser range-finder on a tilt stage, and a stereo camera on a pan-tilt stage. The laser is tilted up and down continuously, providing a 3D view of the area in front of the robot. The resulting point clouds are the input to the perception system, which in turns drives the manipulation system.

### Perception Pipeline

We define a generic framework that can deal with a wide variety of sensors that produce point cloud data. The key characteristics of this perception system are that:

1. It accounts for occlusions correctly by maintaining a model of the environment,
2. It deals with noisy sensor data, especially data obtained from lasers, by removing noise using knowledge of the robot model.

The perception pipeline performs the key task of creating a representation of the world that can be used for collision checking. This representation is updated in realtime, is easily accessible for collision checking and correctly accounts for occlusions. The

[2] `http://www.ada.gov/`

interface to the pipeline is generic in the sense that it can incorporate a wide variety of sensor inputs.

**Sensor Input** The raw input received from the sensor is in the form of a point cloud: a set of points in space that correspond to observed objects in the environment. Most 3D sensors provide information in this format and can be easily plugged into the system described in this chapter. For the implementation on the PR2, the system was interfaced with two different sensors: a Videre stereo camera with projective texture and a tilting Hokuyo laser scanner. The laser sensor is mounted on a tilting stage and it moves up and down at a specified velocity. The viewing angle of the sensor is 270°. This allows the robot to create a detailed representation of the environment in front of it. The stereo camera can provide a denser representation of the environment but was not used as extensively in our implementation.

**Processing Noisy Point Clouds** The sensor data is often noisy and needs to be processed carefully before being incorporated into the robot's view of the world. During manipulation, the arms of the robot are frequently in the sensor field of view. The system must then be able to distinguish sensed points that are coincident with points on the robot itself (see Figure 4.7), i.e., it must be able to infer that sensed points that are on the robot itself are not part of the environment and therefore should not be considered as obstacles.

Such points are separated from the sensor input using a simple approach: for each robot link that could potentially be seen by the robot's sensors, the system checks if any points in the input cloud are contained in the geometric shape corresponding to

the convex hull of that link. This is a simple test and quickly lets the system partition the sensor data into two parts: points that are part of the environment and points that are part of the robot itself and should not be considered obstacles.

An additional problem, called the *shadowing effect*, is especially prevalent in laser scanner data. This problem arises when laser scans slightly graze the different parts of the body of the robot. Points cast by the edges of the arms often appear to be further away and part of the environment. They form a virtual barrier below the arm, on each side, and greatly constrain the motion of the arm. Furthermore, as the arm moves, these shadow points often appear to lie on the desired path of the arm, so execution is halted. To remove these points, a small padding distance is added to the collision representations of the the robot links. If the line segment between a point in the input cloud and the sensor origin intersects the extended collision representation, the point is classified as a shadow point and removed.

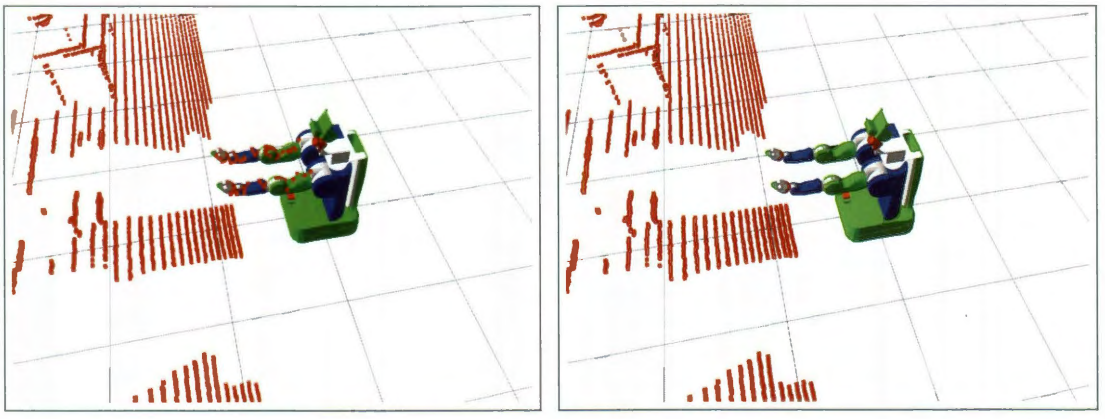


**Figure 4.7:** The robot's world view using its laser without (left) and with (right) filtering.

The filtering process described above is also applied for bodies the robot is manip-

ulating: if the robot is holding an object, that object must not be part of the collision environment any more and the shadow points it casts need to be removed as well. The processed point cloud with shadow points removed can now be processed further for incorporation into the environment representation the robot uses.

**Constructing a Collision Environment** The representation of the collision environment, also referred to as a *collision map*, consists of axis aligned cubes where points from the input cloud are incorporated (see Figure 4.8). Cubes with 1 cm sides were used in the collision map implemented on the PR2. A cubic box is added to the map at a particular location as soon as at least one sensor point is found to occupy the grid cell corresponding to that location. This process is simple and can be executed very quickly.

A proper implementation of a collision environment with frequent sensor updates must deal correctly with occluded data. Replacing the original collision map with only fresh sensor data on every sensor update implies that the map will have no memory about obstacles that may now be occluded. One approach to handling occlusions is to use ray-tracing to trace out every ray coming from the sensors up to a large distance and retain parts of the previous map that are now found to be occluded. This can be very computationally expensive. Since we strive to obtain a perception pipeline that runs close to realtime, we only account for occlusions caused by the robot itself: e.g., when the robot arm is in front of the sensor and parts of the environment are occluded. The simplified approach starts with the previous world representation $C$ (initially empty) and a new world representation $N$. We first determine the set

difference $D$ between the two views, i.e., we look for parts of $C$ that are not part of the new view $N$, i.e.,

$$D = C - N.$$

The parts in $D$ are either moving obstacles that have changed their position or are parts that have become occluded. For every box $d \in D$, we then check whether the line segment between $d$ and the sensor origin intersects a body part of the robot. If it does, the box is considered occluded and is added to the new world view $N$. $N$ now becomes the current representation of the world that retains a memory of the objects seen previously in the environment but now occluded by parts of the robot. This implementation is fast enough to satisfy our requirements for realtime implementation (it runs at around 30Hz - 50Hz with approximately 10000 boxes in the environment).

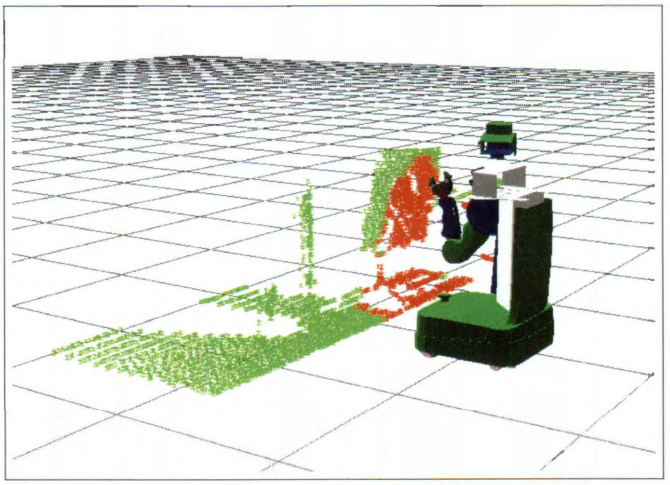


**Figure 4.8:** Example collision map in an office showing retention of occluded data in the environment. Part of the chair is occluded by the arm (marked in red).

The collision map is a critical input to the motion planning and motion execution processes. In our implementation on the PR2 robot, the environment was restricted to a box of size 2m forward, 1.5m on each side and 2m upward, with respect to the robot's base. The box contains the entire reachable workspace of the arm. Restricting the environment size has a significant performance impact and helps in the goal of updating this environment in realtime.

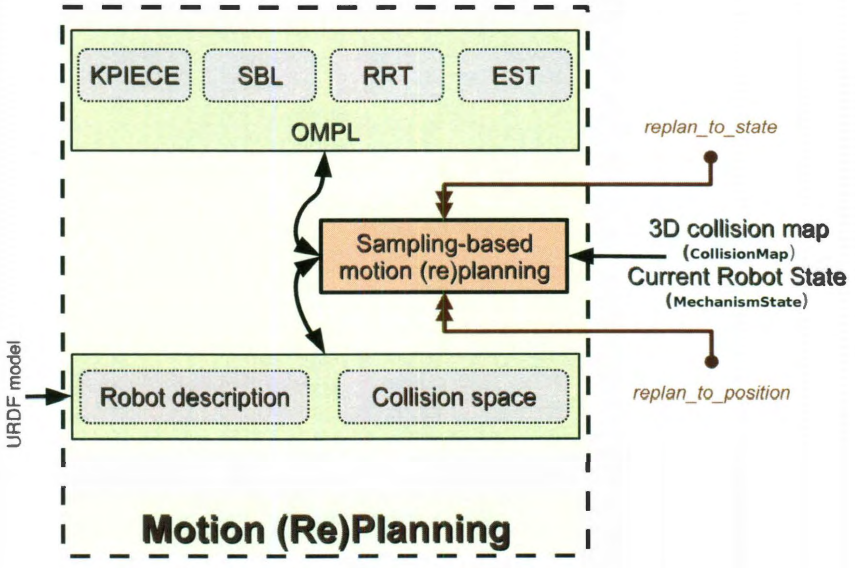



**Figure 4.9:** Diagram of the sampling-based motion planning architecture. Arrows indicate communication between components.

### Motion Planning

The architecture of the used motion planning system is shown in Figure 4.9. Given a URDF description of the PR2, a representation of the state spaces for the arms is

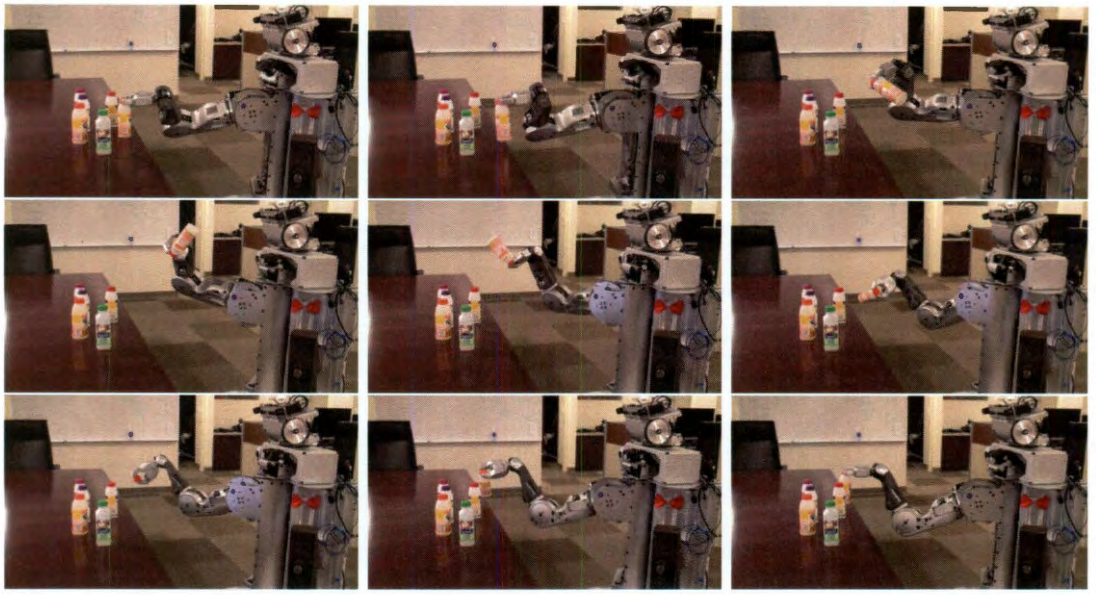


**Figure 4.10:** Example manipulation task that uses OMPL.

automatically constructed for OMPL at runtime.

User requests that require planning to either an arm state (triggered by the "replan_to_state" call) or to an end effector location (triggered by the "replan_to_position" call) are accepted. These requests are converted to OMPL goals: GoalState and GoalRegion, respectively. Planning is then performed with the OMPL library, using the KPIECE algorithm. Because the obtained solution may become invalid in case the environment changes, the execution of a path is continuously monitored. In case of failure, the path is recomputed.

In the practical deployment of the system described above, motion plans were computed at an average of 10Hz. An example application is shown in Figure 4.10. The bottleneck of the system was actually the retrieval of sensor data: the tilting

speed of the laser scanner was low. The development of this system demonstrated the practical applicability of sampling-based planners in real life scenarios, where speed of computation is essential. In particular, the OMPL library and the KPIECE algorithm performed reliably and efficiently in a variety of environments.

# Chapter 5

# Task Motion Multigraphs

This chapter introduces the concept of a *task motion multigraph* (TMM) and shows how TMMs can be used to solve the STAMP problem as defined in Section 1.1.2, in a manner that is more efficient than in previous work. TMMs represent explicitly the state spaces in which motion planning can be performed for a robot to achieve its goal [106, 107]. While TMMs can technically be used with any robotic system, their usefulness is apparent for complex robots, with many degrees of freedom, such as mobile manipulators.

A core issue not captured when representing tasks as graphs (as in Section 1.1.2) rather than TMMs is that there are multiple ways of performing the same operation when the robot used has complex hardware. For example, when asked to reach for an object, a mobile manipulator can use its manipulator alone, it can move its base and then use its manipulator, or it can move both its base and its manipulator simultaneously. More precisely, TMMs represent the possibility that a mobile manipulator

could perform certain tasks using only subsets of its hardware. The experiments included in this chapter show that it is possible to use information from task motion multigraphs and produce fast algorithms that compute sequences of motion plans necessary for solving given tasks.

A recurring issue with task and motion planning is that of computation time. For many practical applications, reduced computation times are desirable. For instance, a robot performing tasks in a changing environment must be able to recompute its tasks quickly in order to react to observed changes. In addition, avoiding unnecessary motions is desirable (e.g., while the robot's base is moving, the robot's arms should stay fixed if possible). This work proposes a solution for the above issues, in the context of mobile manipulation, under some assumptions that are mentioned later on.

Again of practical interest is the consideration of uncertainty, be that caused by the robot's inability to perfectly execute specified commands, imperfections in the robot's perception, or other sources. Chapter 6 covers this aspect as well, again in the context of mobile manipulation, using TMMs.

This chapter continues by stating the considered problem setup in Section 5.1 and then showing how TMMs can be constructed in Section 5.2. An algorithm that uses TMMs is described in Sections 5.3 and 5.4. Finally, experimental results are presented and discussed in Section 5.5.

## 5.1 Problem Scenario

**Assumptions** We consider a single robotic device. This assumption is made for the simplification of the prose and of the experiments, rather than for theoretical reasons.

We assume that a task specification is available, in any of the variants suggested by previous work (LTL [18], STRIPS-like [2], etc.). Such specifications can be used to construct explicit task graphs (e.g., using techniques from artificial intelligence [2]). In some cases, the explicit construction is possible only if the horizon of actions is bounded. This is a reasonable assumption for robots such as mobile manipulators operating in human environments. Task graphs can also be specified explicitly (e.g. [17]) and in that case a higher-level specification is not needed. For simplicity, assume the only actions the robot can perform are `grip` (close gripper), `release` (open gripper) and `move_to` (plan a motion). The `grip` and `release` actions are very simple ones and do not include the computation of grasp poses. It is assumed that if grasp poses are necessary (which is typically the case), a grasp reasoning system (e.g., [70]) is employed at the time the task graph is generated and grasp poses are included in the graph's nodes. It is further assumed that such grasp poses, when specified, can be converted to states, or sets of states. Such computation can be performed, for instance, with inverse kinematics [108, 109].

**Robotic Devices** From a theoretical standpoint, TMMs can be used with any robotic system. From a practical point of view, use of TMMs is beneficial for robots with many degrees of freedom, such as mobile manipulators, which are often capable of performing their tasks using subsets of their available hardware components.

Mobile manipulators are robotic devices that include a means of locomotion (e.g., tracks, wheels, legs) and a means of interaction with the environment (usually arms). Mobile manipulators are often complex, with many degrees of freedom (e.g., Honda Asimo, Willow Garage PR2; see Figure 5.1). Their complexity makes them versatile, capable of solving a variety of tasks without undergoing hardware changes specific for particular tasks. In fact, given a specific task, it is possible a mobile manipulator can perform that task in a multitude of ways, depending on its choice of hardware components. For example, a mobile manipulator can open a door by simply extending its arm and pressing the door's handle. At the same time, it is possible for the robot to get closer to the door by moving its base, and then pressing the handle with its arm. Furthermore, it is possible for the robot to use its arm and base simultaneously.

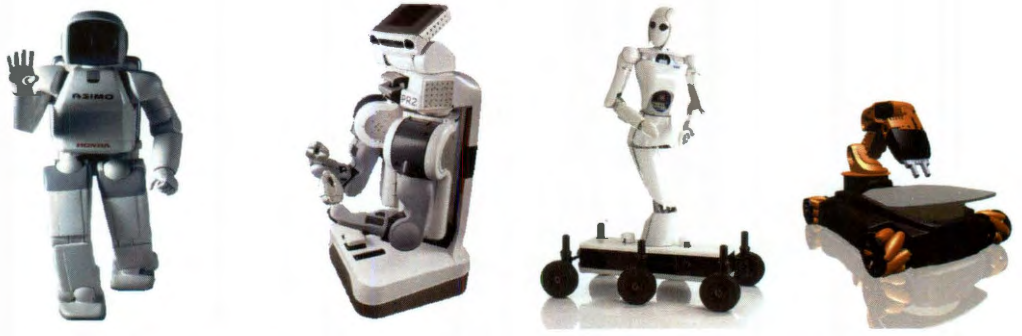



**Figure 5.1:** Examples of mobile manipulators (from left to right): Honda Asimo, Willow Garage PR2, DFKI AILA, KUKA youBot.

In this chapter, robotic devices are viewed simply as sets of joints $J$. For example, the PR2 mobile manipulator consists of the joints in its base, torso, arms and head. Furthermore, we consider an additional joint that connects the robot to the

environment. For an omni-directional robot moving in plane, such as the PR2, this joint corresponds to 3 DOF (position and rotation in plane) – an SE(2) joint. A set of joints $J$ implicitly defines a state space $\mathcal{X}_J$. Often, groups of joints are controlled simultaneously, typically when the joints make up a functional part of the robot's hardware. Typical examples of groups of joints that are controlled together are the joints in the base and joints in the arm(s).

## 5.2 Construction and Definition of TMMs

This section shows how a TMM is constructed from a problem specification (a task graph) and information about the robot hardware to be used. First, an intermediate notion is introduced, *the task motion graph* (TMG), and then a definition of TMMs is given.

### 5.2.1 Task Motion Graphs

Figure 5.2-Left shows an example task graph. This example encodes the task of delivering a book, while accounting for the possibility of having to move a cup of coffee out of the way, if delivering the book directly is not possible. The `move_to` actions are of special interest since these are the ones that require motion planning. Contracting the edges that correspond to `grip` and `release` actions in the task graph leads to a simplification as shown in Figure 5.2-Right. We refer to this simplification as the *task motion graph.* This does not mean that the `grip` and `release` actions are not going to be performed in the execution of the task plan. Removing these edges

is only a simplification that allows us to focus on the motion planning actions. The TMG also encodes the different state spaces that could be used for motion planning, indicated as labels on edges. What these state spaces are, depends on the hardware characteristics of the considered robotic system. The example in Figure 5.2-Right is for a mobile robot with one arm. The state spaces used correspond to sets of joints that are assumed to be controlled together: the arm and the base. Essentially, the TMG encodes the different sequences of motions that could take the robot to its goal, accounting for the hardware components that could possibly be used to execute those motions.

For convenience, we remind the reader that the mobile manipulator consists of a set of joints $J$ and this implicitly defines a state space $\mathcal{X}_J$. Furthermore, a task graph is assumed to be defined as in Section 1.1.2.

**Definition 5.2.1** Task Motion Graphs.

*A task motion graph (TMG) for mobile manipulation is a directed acyclic graph* $G = (V, E)$ *such that:*

- $V = \{v \mid Q(v) \subset \mathcal{X}_J\}$. *Every vertex* $v$ *is associated with a set of states* $Q(v) \subset \mathcal{X}_J$. $Q(v)$ *can be explicitly specified as a set of states or implicitly specified in a manner that allows computation of states in* $Q(v)$ *(e.g., end-effector poses, which can be converted to states using inverse kinematics [108]).*

- $E \subset V \times V$ *such that* $\forall v \in V, (v, v) \notin E$ *and there exists an edge labeling function* $label(e) = (Act, Env_e, \mathbf{A}_e)$. *The existence of an edge* $(v_1, v_2) \in E$ *implies there exists an action that can take the robot from* $v_1$ *to* $v_2$*, and that action requires motion planning. Act specifies the action that requires motion planning. For the*

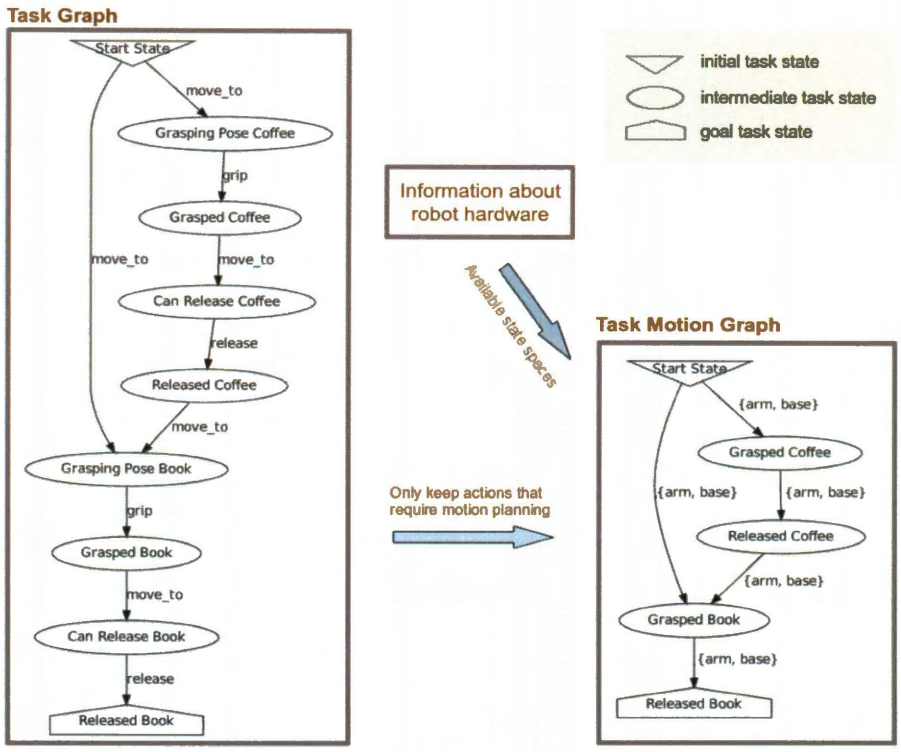



**Figure 5.2:** Left: The task graph for delivering a book. It may be necessary for a cup of coffee to be moved out of the way to deliver the book. Right: The task motion graph – only actions that require motion planning are kept, and they are labeled with the state spaces that could possibly be used for motion planning.

*purposes of this work, Act is always* `move_to` *(Act* $\in \mathcal{T}$ *as in Section 1.1.2). At the start of the action, the robot is at a state* $x \in Q(v_i)$ *and at the end of the action, the robot is at a state* $x' \in Q(v_j)$*. Env*$_e$ *defines the environment in which motion plans for edge* $e$ *are to be computed.* $\mathbf{A}_e = \{A_{e,1}, \ldots, A_{e,k} | A_{e,\cdot} \subseteq J, k < 2^{|J|}\}$ *defines the possible sets of joints to plan for when computing motion plans along edge* $e$*.*

- $root \in V$ *and* $Q(root)$ *consists of a single element: the starting state of the robotic system.*

- $F \subseteq V$, $F \neq \emptyset$ *is the set of goal states for the task.*

It is important to note that there can be multiple sets of joints that can be used when computing motion plans along an edge $e$. $\mathbf{A}_e$ is an input specified by the user. For example, if moving an arm's end-effector is intended, $A_{e,1}$ can be defined to be the minimal set of joints usually required to perform the operation: the joints in the actual arm. For more complex problems, it may be necessary to move the robot's base as well. For this reason, additional sets of joints ($A_{e,2}, \ldots$) can be included in $\mathbf{A}_e$. The intention is that the robot can use any combination of its available hardware components, which corresponds to planning in any state space $\mathcal{X}_L$ (orthogonal projection of $\mathcal{X}_J$), for $L = \bigcup_{j \in A} j$, $A \subseteq \mathbf{A}_e$. From a practical standpoint, the choice of $\mathbf{A}_e$ also depends on what controllers are available because elements of $\mathbf{A}_e$ (sets of joints) typically correspond to the sets of joints that can be controlled together.

*Example:* Consider a mobile manipulator with an arm with 7 joints and an omni-directional base that moves in plane. Assume the arm and the base can be controlled independently. We define $J_{arm}$ to be the set of joints in the arm and $J_{base}$ to be a virtual joint with 3 degrees of freedom that corresponds to the $SE(2)$ state space ($\mathcal{X}_{J_{base}} = SE(2)$). We thus have $\mathcal{X}_{J_{arm}}$ 7-dimensional, $\mathcal{X}_{J_{base}}$ 3-dimensional and $\mathcal{X}_J$ 10-dimensional, $J = J_{arm} \cup J_{base}$. A simple TMG $G = (V, E)$ can have $V = \{v_{start}, v_{goal}\}$, $Q(v_{start}) = \{x_{start}\}$, $Q(v_{goal}) = \{x_{goal}\}$, defines the two vertices of the TMG: a start state and a goal state. $E = \{e = (v_{start}, v_{goal})\}$, $label(e) = (\texttt{move\_to}, Env, \{J_{arm}, J_{base}\})$, $root = v_{start}$, $F = \{v_{goal}\}$. This specification defines

a TMG that requires the computation of a motion plan from a specified start state $x_{start} \in \mathcal{X}_J$ to a state $x_{goal} \in \mathcal{X}_J$. The motion plan is to be computed in either $\mathcal{X}_{J_{arm}}$, $\mathcal{X}_{J_{base}}$ or $\mathcal{X}_{J_{arm} \cup J_{base}}$. Planning may not be feasible for all possible combinations of spaces specified by $\mathbf{A}_e$. *Env* represents the environment considered for determining the validity of states (e.g., collision checking).

### 5.2.2 Task Motion Multigraphs

A *task motion multigraph* (TMM) is the explicit representation of a TMG, in a manner that can be used for motion planning.

**Definition 5.2.2** Task Motion Multigraphs.

*Given a TMG, $G = (V, E)$, we define a task motion multigraph (TMM), $G_M = (V_M, E_M)$ as follows:*

- $V_M = V$.
- *for every $e = (v_i, v_j) \in E$, $label(e) = (Act, Env_e, \mathbf{A}_e)$, let $E_{M,e}$ be a multiset, $E_{M,e} = \{e_{m,k} = (v_i, v_j) \mid k \in \{1, \ldots, 2^{|\mathbf{A}_e|} - 1\}\}$ and $label_M(e_{m,k}) = (Act, Env_e, J_{e,k})$, for $J_{e,k} = \bigcup_{j \in \mathbf{a}(k)} j$, $\mathbf{a} : \{1, \ldots, 2^{|\mathbf{A}_e|} - 1\} \rightarrow 2^{\mathbf{A}_e} \setminus \{\emptyset\}$ is a bijection, $2^{\mathbf{A}_e}$ is the power set of $\mathbf{A}_e$. $E_M = \bigcup_{e \in E} E_{M,e}$.*
- *$root_M \in V_M$ corresponds to $root \in V$.*
- *$F_M \subseteq V_M$, $F_M \neq \emptyset$ corresponds to $F \subseteq V$.*

In essence, a TMM is a TMG where all possible sets of joints used in motion planning are explicitly specified for each edge. The conversion, which can be done

automatically, is fairly straightforward and only requires addition of edges. See Figure 5.3 for an example. The TMM reveals an additional layer of complexity for mobile manipulation: the need to decide which state spaces to plan in. This is a different type of decision to be made, in addition to other decisions such as selection of grasping poses. We introduce TMMs in an attempt to expose the need to consider which state spaces to plan in. This observation has been made in previous work as well [10, 12], but this work presents the first formalization. Since there is a combinatorial explosion in the construction of a TMM from a TMG (in terms of number of edges), it is desirable that the TMG is defined in a manner amenable to the robotic system this notion is used for. For instance, a robotic system with a mobile base and two arms may define three sets of joints to be used for planning: $J_{left}$, $J_{right}$, $J_{base}$, corresponding to the joints in the respective arms and the base. Edges in the TMG could then use $\mathbf{A}_e = \{J_{left}, J_{right}, J_{base}\}$. The definition of $\mathbf{A}_e$ implies that the option of planning in the full state space, $\mathcal{X}_{J_{left} \cup J_{right} \cup J_{base}}$, is included in the edges of the TMM, allowing in this case even the use of control theoretic techniques, if available [110].

**Definition 5.2.3** A TMM plan (A motion plan for a TMM).

*A TMM plan for a TMM $G = (V, E)$ is an ordered sequence of edges $P = \{e_1, \ldots, e_k\}, P \subseteq E$ such that for every edge $e = (v_a, v_b) \in P$ there exists a motion plan between some state $x_a \in Q(v_a)$ and some state $x_b \in Q(v_b)$. Furthermore, the motion plans for any two consecutive edges $e_i, e_{i+1}, 1 \leq i < k$ from $P$ can be* connected. *The two motion plans are said to be connected if there exists a well-defined method (e.g., a controller) to move from the last state $x_{i,L}$ of the motion plan for $e_i$ to the first state $x_{i+1,B}$ of the motion plan for $e_{i+1}$. For the purposes of this work, the*

*condition* $x_{i,L} = x_{i+1,B}$ *was imposed for connectivity to be achieved.*

*A TMM plan* $P = \{e_1, \ldots, e_k\}$ *is a solution in a TMM if* $e_1 = (root, \cdot)$ *and* $v_b \in F$ *($v_b$ is a goal), with* $e_k = (v_a, v_b)$.

*Remark:* Given a TMM plan, it is easy to see that a task plan in the original task graph can be constructed: the actions from the task graph that are not present in the TMM do not require motion planning. Only `grip` and `release` actions (closing and opening the end-effector) need to be re-inserted.

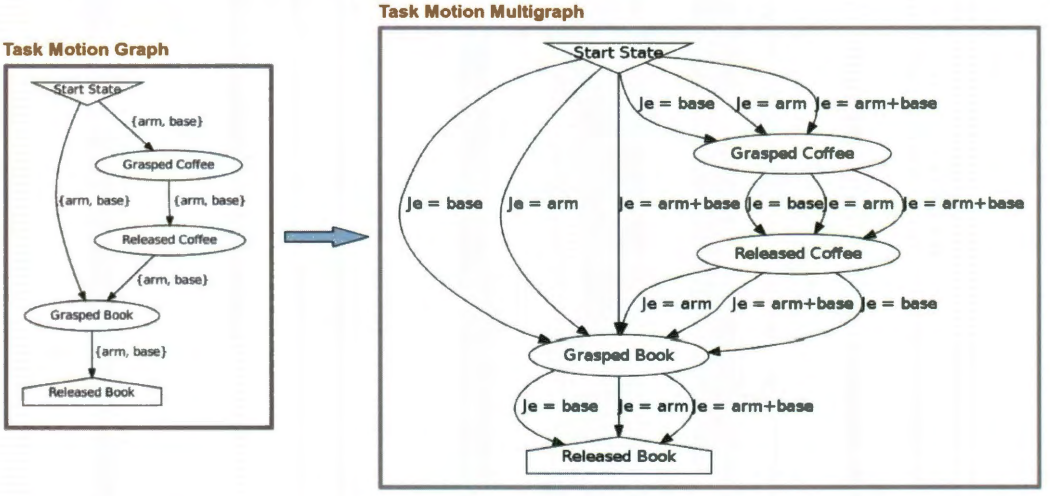



**Figure 5.3:** Left: The task motion graph for delivering a book, defining the groups of joints $\mathbf{A}_e$. Light: The task motion multigraph. Edges define $J_e$.

## 5.3 Task and Motion Planning with TMMs

**Intended Use** The intended use scenario of task motion multigraphs is as follows. A task graph with bounded horizon is provided as input. This work does not concern

itself with how the task graph is constructed: its existence is assumed (e.g., [17]). A corresponding TMG is then constructed by: *(1)* discarding the actions that do not require motion planning and *(2)* attaching information to the TMG edges about possible state spaces to plan in. The TMG is then converted to a TMM, and a TMM plan that solves the STAMP problem is computed.

This chapter provides a formalization of the available motion planning options in a manner that can facilitate computation. The following section shows an example method that uses information contained in the TMM to compute motion plans. The availability of the TMM at the time of motion plan computation helps with the identification of less expensive but feasible sequences of motion plans. This information is used to attempt to reduce the amount of time spent planning motions.

**Baseline Algorithm** Computing motions plans for certain edges in the TMM may be more time consuming than for others: the complexity of the environments can vary, the set of joints $J_e$ to plan for (and implicitly, the dimensionality of $\mathcal{X}_{J_e}$) may also vary. Planning motions along some edges may not even be feasible. These considerations make it obvious that computing the sequence of motion plans for some paths in the TMM can be much more computationally intensive than for others. To address this issue, we attempt to compute motion plans for the path that appears to be the cheapest. Which path appears to be the cheapest changes as the computation progresses. The cost of a path is the sum of the costs of its edges. The cost of an

edge $e$, $label(e) = (Act, Env_e, J_e)$ is:

$$cost(e) = \exp\left(\frac{dim(\mathcal{X}_{J_e})}{\max_J dim(\mathcal{X}_J)}\right) \cdot \begin{cases} 1 & \text{if } sol \\ s \cdot (1+t) \cdot \left(1 + \frac{d_L(e)}{d_R(e)+d_L(e)}\right) & \text{if not } sol, \end{cases}$$

where $dim(\mathcal{X}_{J_e})$ is the dimension of the state space to be used for planning along edge $e$, $\max_J dim(\mathcal{X}_J)$ is the dimension of the largest state space considered by the TMM, $s$ represents the number of times $e$ was selected for motion planning (starts at 1), $t$ is the number of seconds already spent planning motions along $e$, $d_L(e)$ represents the number of edges from $e$ to the closest goal vertex, and $d_R(e)$ represents the number of edges from $e$ to the root. If motion plans along edge $e$ are already available (*sol* is true), the cost of $e$ relates only to the dimensionality of the state space, thus making edges that actuate fewer joints preferable. If no motion plans for $e$ are available (*sol* is false), the cost of $e$ is increased proportionally to the number of times $e$ was selected for planning. Furthermore, the closer $e$ is to a possible goal, the fraction $d_L(e)/(d_R(e) + d_L(e))$ decreases, thus decreasing the cost of edges closer to possible goals.

The definition for the cost of an edge is heuristically determined, with the purpose of approximating how expensive paths are. The provided formula is intended as a guide and represents what worked well in the presented experiments. In general, the dimensionality of the space to plan in and the amount of time spent planning seem to be the more important parameters when estimating edge costs. Determining better edge costs is an open issue that can affect performance. This is a topic for further investigation and several other approaches may be applicable (e.g., [73]).

The computation of a task plan in a TMM proceeds as described in Algorithm 4. The body of the algorithm is an iterative process that runs motion planners on different edges of the TMM for short periods of time. There are two main steps in the iteration. The first step [lines 2-4] aims to find motion plans for TMM edges that are closer to the goal (greedily selected). If no progress towards the task goal is made, the second step [lines 5-7] is executed. The second step aims to find motion plans for TMM edges selected stochastically, so that probabilistic completeness can be achieved.

At every iteration, the set of segments that make up the path of least cost from the root to a goal node in $G_M$ is computed using Dijkstra's algorithm [line 2]. The edge along this path that is closest to the goal and that has no motion plan associated to it is then selected deterministically by *SelectEdgeFromPath()* [line 3]. Such an edge will exist as long as no complete solution has been found for the TMM. Motion planning is performed on the selected edge for a short duration ($\Delta t$) [line 4]. Repeated selections of the same edge continue the computation of the motion plan rather than restart it. Unless a motion corresponding to the selected edge is successfully computed, motion planning is executed on another edge, one selected from the remaining set of edges in the TMM [lines 6, 7]. With low probability (10% in this work) *SelectEdge()* selects an edge randomly; otherwise, priority is given to edges that have fewest number of computed motions, then to edges that have lowest cost. The intention is to be greedy and follow the path of least cost to the goal but at the same time allow the exploration of other options. Planning is performed for short durations $\Delta t$ to allow updating edge costs more often.

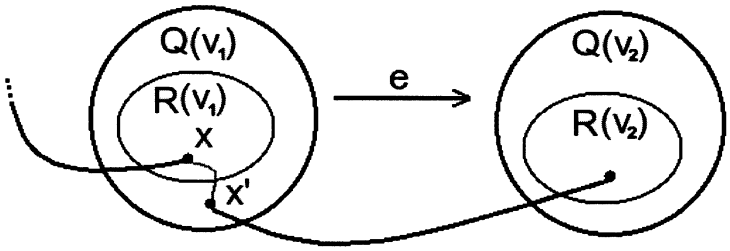



**Figure 5.4:** Diagram showing a computed motion along edge $e = (v_1, v_2)$. The motion starts at state $x' \in Q(v_1)$, reaches a state in $Q_R(v_2)$ and there exists a means to connect to $x'$ from $x \in Q_R(v_1)$.

**Algorithm 4** TMM-Computation($G_M = (V_M, E_M)$)

```
1: while timeSpent < MaxT do
2:   P ← ShortestPath(G_M)
3:   edge ← SelectEdgeFromPath(P)
4:   sol ← MotionPlan(edge, Δt)
5:   if sol = nil then
6:     nextEdge ← SelectEdge(E_M \ M(edge))
7:     sol ← MotionPlan(nextEdge, Δt)
8:   if sol ≠ nil then
9:     RecordSolution(edge, sol)
10:  if HaveSolution(G_M) then
11:    return ExtractSolution(G_M)
12: return nil
```

For a vertex $v \in V$, let the reached states in $Q(v)$ be denoted by $Q_R(v) \subseteq Q(v)$. A state $x$ is reached if there exists a TMM plan that ends at $x$ (see Definition 5.2.3). Initially, $Q_R(v) = \emptyset$ for all vertices, except for the root: $Q_R(root) = Q(root)$. For a new motion plan to be computed for an edge $e = (v_1, v_2)$, a starting state needs to be identified in $Q(v_1)$. A state $x' \in Q(v_1)$ can be used as a starting state for a motion plan along $e$ if some state $x \in Q_R(v_1)$ has been previously reached by a TMM plan, and $x$ can be connected to $x'$ using a well-defined means (in this work, $x = x'$).

When a motion plan for an edge $e$ is computed, the reached state in $Q(v_2)$ is added to $Q_R(v_2)$ (via *RecordSolution()*). A diagram showing the computation of a motion is in Figure 5.4. If $Q_R(v_1) = \emptyset$, no motion plan is computed for $e$. This is done to avoid computing motion plans that start at unreachable states.

Termination for the algorithm is possible in two ways: either $HaveSolution()$ returns true [line 12] and a solution is found, or the total time spent planning exceeds $MaxT$, in which case the algorithm returns failure.

## 5.4 Sharing Exploration Information with TMMs

In this section, we show how to further leverage the information contained in the TMM to decrease computational effort. The additional idea is that when planning in higher-dimensional spaces is required to find solutions, information from the exploration of lower-dimensional spaces is reused. Samples generated while exploring lower-dimensional projections $\mathcal{X}_{J''}$ are lifted to the full state space of the robot and to higher-dimensional projections $\mathcal{X}_{J'}$, $J'' \subseteq J'$. Thus, when reverting to planning in higher-dimensional spaces, the motion planner does not start from scratch. An additional feature of our approach is that it can implicitly perform decoupled planning [4], as discussed later. This idea is developed for planning under geometric constraints.

**Preliminaries** Let $G_M = (V_M, E_M)$ be the TMM given as input. For every edge $e = (v_a, v_b) \in E_M$, $label(e) = (Act, Env, J_e)$, define the following operators:

- $M(e) = \{(v'_a, v'_b) \in E_M \mid v_a = v'_a, v_b = v'_b\}$, the multi-set of edges that connect

the same pair of nodes as edge $e$,

- $Joints(e) = J_e$, the set of joints $e$ corresponds to (specified by $label(e) = \{Act, Env_e, J_e\}$),
- $Space(e) = \mathcal{X}_{J_e}$, the state space $J_e$ implicitly defines.

Let $\mathcal{X}_J$ be the full state space of the robot. Given $x \in \mathcal{X}_J$ and $y \in \mathcal{X}_{J'}$, $J' \subset J$, let $Lift(y, x) \in \mathcal{X}_J$ be the state that has the same values as $y$ for the joints in $J'$ and the same values as $x$ for the joints in $J \backslash J'$. The inverse operation of $Lift()$ is $Project()$. For $x \in \mathcal{X}_J$, $Project(x, \mathcal{X}_{J'}) = y \in \mathcal{X}_{J'}$, where $y$ has the same values as $x$ for the joints in $J'$ ($y$ is an orthogonal projection of $x$).

**Updated TMM Algorithm** Algorithm 5 shows an updated version of Algorithm 4. The difference with respect to Algorithm 4 is that instead of directly calling a motion planner (*MotionPlan()*, lines [4, 7]), a more complex routine is called (*TMM-MotionPlan()*, shown in Algorithm 6). To perform its computation, *TMM-MotionPlan()*, just like *MotionPlan()*, receives as arguments the edge (*edge*) to plan motions for and an amount of time to spend planning ($\Delta t$). The difference is that *TMM-MotionPlan()* may need to switch to a different space to plan in, one that corresponds to an edge in $M(edge)$. To accommodate this change, *TMM-MotionPlan()* returns the edge it computed a motion plan for, in addition to the actual motion plan (if one is computed).

Given a TMM edge $edge = (v_a, v_b)$ and an amount of time $\Delta t$, Algorithm 6 computes a valid motion between $v_a$ and $v_b$ along a TMM edge in $M(edge)$ within

**Algorithm 5** TMM-Computation($G_M = (V_M, E_M)$)

```
1: while timeSpent < MaxT do
2:   P ← ShortestPath(G_M)
3:   edge ← SelectEdgeFromPath(P)
4:   (edge', sol) ← TMM-MotionPlan(edge, Δt)
5:   if sol = nil then
6:     nextEdge ← SelectEdge(E_M \ M(edge))
7:     (edge', sol) ← TMM-MotionPlan(nextEdge, Δt)
8:   if sol ≠ nil then
9:     RecordSolution(edge', sol)
10:  if HaveSolution(G_M) then
11:    return ExtractSolution(G_M)
12: return nil
```

the amount of time $\Delta t$, or terminates with failure. The availability of a bi-directional motion planning algorithm is assumed. An instance of such an algorithm (*edge*.mp) and storage for its generated exploration information are associated to every edge in the TMM. Sampling-based planners would be typically used for *edge*.mp, but the only needs for an algorithm to be usable are that it must allow: *(1)* access to the valid motion segments it generates in its exploration (*readNextValidMotionSegment()* function used in Algorithm 6), and *(2)* a means of incorporating information about new valid motion segments that are computed externally (*addValidMotionSegment()* function used in Algorithm 6). The method about to be described could be used with uni-directional motion planners as well, but the resulting implementation would be less efficient.

Algorithm 6 manages the exploration information generated by the motion planning instances associated to the TMM's edges. Significant computational gains can be obtained by sharing exploration information between the planning instances. The

overall memory consumption of our approach is not affected negatively, as will be shown later. This approach requires essentially no changes to the underlying motion planner: the sharing of exploration information is managed completely by Algorithm 6.

The first time *TMM-MotionPlan()* is called for *edge*, input states are added for all edges in $M(edge)$ [lines 1-3 Algorithm 6, Algorithm 7]. *ActivatePlanner()* starts the motion planner corresponding to *edge*, and stops any other running motion planner instance, if one is active [line 4]. The motion planner is automatically deactivated when *TMM-MotionPlan()* terminates. It is assumed that the activated motion planner runs in background, while Algorithm 6 executes. However, the implementation used in the experimental section is single-threaded, so that the benefits of the proposed algorithm are exposed, rather than the benefits of increased computational power.

The body of Algorithm 6 is a three part iterative process. At every iteration, the *readNextValidMotionSegment()* function is called [line 5] to obtain a pair of states ($x_p$, $x$) that represent a new valid motion segment discovered by the motion planner in use. Information gained at each iteration is propagated to higher-dimensional spaces in the first part of the algorithm. When solutions are found in lower-dimensional spaces, part two decides whether to report a solution or to switch to planning in different state spaces. Part three switches to planning in higher-dimensional spaces if slow progress is detected. More details on these parts follow.

*1) Part one* of Algorithm 6 [lines 6-12] shares information gained from the exploration of $Space(edge)$ with motion planners that could potentially be called for other

**Algorithm 6** TMM-MotionPlan($edge = (v_a, v_b)$, $\Delta t$)

1: **if not** AddedInputStates($edge$) **then**
2: TMM-AddInputStates($edge$, $Q_R(v_a)$, $Q(v_b)$)
3: ActivatePlanner($edge$.mp)
4: **while** $timeSpent < \Delta t$ **do**
5: $(x_p, x) \leftarrow edge$.mp.readNextValidMotionSegment()

**Part 1**: // share information between planning spaces

6: **if** $x \neq$ `nil` **then**
7: **for** $e' \in M(edge)$ **do**
8: **if** $Joints(edge) \subset Joints(e')$ **then**
9: $spc \leftarrow Space(e')$
10: $y_p \leftarrow Project(Lift(x_p, Root(x_p)), spc)$
11: $y \leftarrow Project(Lift(x, Root(x)), spc)$
12: $e'$.mp.addValidMotionSegment($y_p$, $y$)

**Part 2**: // report solution or plan for remaining dimensions

13: **if** $edge$.mp.haveSolution() **then**
14: $(sol, x_c) \leftarrow edge$.mp.getSolution()
15: **if** $FullDimensionalSolution(sol)$ **then**
16: **return** $(edge, sol)$
17: $edge$.used $\leftarrow$ True
18: $e' \leftarrow$ NextPlanningEdge($M(edge)$)
19: $x_c^s \leftarrow Lift(x_c, StartRoot(x_c))$
20: $x_c^g \leftarrow Lift(x_c, GoalRoot(x_c))$
21: TMM-AddInputStates($e'$, $\{x_c^s\}$, $\{x_c^g\}$)
22: **return** TMM-MotionPlan($e'$, $\Delta t - timeSpent$)

**Part 3**: // if slow progress, switch to higher-dimensional spaces

23: **if** SlowProgress() **and** $Space(edge) \neq \mathcal{X}_J$ **then**
24: $edge$.used $\leftarrow$ True
25: **for** $e' \in M(edge)$ **do**
26: **if** $Joints(edge) \subset Joints(e')$ **then**
27: **return** TMM-MotionPlan($e'$, $\Delta t - timeSpent$)

28: **return** `nil`

**Algorithm 7** TMM-AddInputStates($edge$, $s$, $g$)

1: **for** $e' \in M(edge)$ **do**
2: $e'$.mp.addInputStates($Project(s, Space(e'))$, $Project(g, Space(e'))$)

edges in $M(edge)$. The goal is to reuse information between planning instances, so that if planning in higher-dimensional spaces is needed, the planner does not start from scratch.

If the motion segment between states $x_p \in Space(edge)$ and $x \in Space(edge)$ is valid, an equivalent motion segment from $y_p \in Space(e')$ to $y \in Space(e')$ can be constructed for some edges $e' \in M(edge)$. The condition on edges $e'$ is that $Joints(edge) \subset Joints(e')$. For example, for the TMM in Figure 5.2-Right, exploration in the space $\mathcal{X}_{J_{arm}}$ would lead to progress in the state space $\mathcal{X}_{J_{arm+base}}$ as well. To compute the equivalent motion segment, the states $x_p$ and $x$ first need to be lifted to the full state space of the robot, $\mathcal{X}_J$. This can always be done because plans from the original input state to $x_p$ and to $x$ are known. All joint values are known for the original input state, so states $x_p$ and $x$ can be lifted to $\mathcal{X}_J$ by filling in the missing joint values with the ones from the input states. The lifted states can then be projected to $Space(e')$, yielding a valid motion segment that can be added to the motion planner instance exploring $Space(e')$ [lines 9-12]. In the worst case scenario, there could be $2^{|\mathbf{J}|-1} - 1$ sets $J'$ such that $Joints(edge) \subset J'$. While the number of state spaces to keep track of is exponential, the validity of a motion (collision checking) is evaluated only once. Furthermore, when the validity check is performed, all the robot parts need to be checked for collision, and the full robot state is actually al-

ready constructed. The process of lifting and projecting that state can be made very efficient.

*2) Part two* of Algorithm 6 [lines 13-22] handles the construction of solution plans and the possibility of decoupled planning. When a solution is found in $Space(edge)$, it is possible that not all of the joints for the input start and goal states are matched, since a plan was found only for a subset of the joints of the robot (e.g., a plan for the base only, will not ensure that the arm has moved to its correct state). In that case, planning is subsequently attempted for a subset of the unmatched set of joints.

The *FullDimensionalSolution()* routine checks if the obtained solution covers all the dimensions of $\mathcal{X}_J$ [line 15]. If $Space(edge) = \mathcal{X}_J$, *FullDimensionalSolution()* will return true. If an incomplete solution is found, more planning needs to be done, perhaps in a different state space. The *NextPlanningEdge()* routine decides which edge to switch to. A constraint at this point is that no edge is active more than once (mechanism implemented with the *edge*.used variable) to avoid infinite recursion. *NextPlanningEdge()* identifies a TMM edge $e'$ whose corresponding joint values differ between the start and goal states, such that the dimension of $Space(e')$ is minimal and $e'$.used is false). For example, if planning for the base, left arm and right arm, it may be necessary to switch to the space corresponding to the left arm after having succeeded at planning a motion for the base alone. This is because even though an SE(2) plan for the robot's base was found, the arm may not be at the desired state. In order to reuse the incomplete solution found while planning in $Space(edge)$, additional input states are added [line 21]. Because the motion planner we use is bi-directional, a connection state $x_c \in Space(edge)$ along the solution exists such that $x_c$ is connected

to both a starting state and a goal state. The state $x_c$ can be lifted to $\mathcal{X}_J$ in two ways, using either of the input states it is connected to [lines 19,20]. Although $x_c^s \neq x_c^g$, they do not differ for the joints in $Joints(edge)$. Subsequent planning in $Space(e')$ may quickly lead to a solution. This is in fact a form of decoupled planning [3,4]. For example, a motion for the base could be planned first, and a motion for the arm could be planned subsequently. This approach may lead to faster computation of solutions, but it is not a complete approach.

*3) Part three* of Algorithm 6 ensures that *TMM-MotionPlan()* eventually degrades to simply calling the motion planner for $\mathcal{X}_J$ [lines 23-27]. If slow progress is detected, a switch is made to a strictly higher-dimensional space that requires planning for a larger set of joints. The condition we use for detecting slow progress is that the distance between the set of states connected to starting states and the set of states connected to goal sates does not decrease for two thousand iterations. Due to this degradation policy, if the underlying motion planner is (probabilistically) complete, the same property is maintained for *TMM-MotionPlan()*.

## 5.5 Experimental Results

The benefits of using TMMs are experimentally evaluated in the context of motion planning under geometric constraints. Four environments are defined: "Office1" in Figure 5.6, "Office2" in Figure 5.8, "Office3" in Figure 5.10 and "Winding Tunnels" in Figure 5.12. Each of the figures consists of two sides: the left side is an image of the environment and the right side is the TMG to be used. The leafs of the TMGs are

assumed to be the goal regions. The robot considered is the PR2 from Willow Garage. A black dot at the bottom-left of each of the figures indicates the robot's scale in the environment. The sets of robot joints used for planning are the left arm (7 DOF), the right arm (7 DOF) and the base (3 DOF) – a total of 17 DOF. Thus, the label of every edge in the TMG is {base, left_arm, right_arm}, and seven $(2^3 - 1)$ edges are constructed in the TMM for every TMG edge. This represents the worst case scenario, as not all edges would be necessary in practical applications. For example, the edge corresponding to the left and right arms is usually unnecessary.

"Office1" and "Office2" are relatively simple office-like environments, with "Office1" being a bit more cluttered. "Office3" is a more complex office-like environment due to the highly constrained regions that are required to be reached. Furthermore, the regions marked in Figure 5.10-Left always differ in all the robot's joints, so planning for all 17 joints is always necessary. "Winding Tunnels" is a complex environment as well, requiring the navigation of narrow hallways.

RRT-Connect [41] was used as the underlying planner in Algorithm 4 and Algorithm 6 because it is a simple and well established algorithm. However, other algorithms could be used as well. Planning is done under geometric constraints only and the implementation used is from OMPL [93].

This section shows experimental results for solving the task and motion planning problem in three ways:

1. Using Algorithm 5 (which calls Algorithm 6 internally) and supplying a TMM as input, for the problems described above. This method is denoted as "TMM2".

2. Using Algorithm 4 and supplying a TMM as input, for the problems described above. This method is denoted as "TMM1".

3. Using Algorithm 4 and supplying graphs as input rather than TMMs (the edges in the input graph always correspond to planning in the full state space of the robot). This method is denoted as "graph".

Although the implementation of our TMM-based approach is easily parallelizable, we use a single threaded implementation in our experiments. The total execution time allowed for the algorithms ($MaxT$) is 600 seconds.

**Sample Execution** Before presenting detailed experimental results, a sample execution of Algorithm 5 on the "Office1" environment is shown in Figure 5.5. Because the actual TMM is too large to display, the TMG is shown at each step, with bits of additional information from the execution of the algorithm.

The first step of the execution shows the input TMG. Each of the following steps indicates the currently considered task solution: the path with bold edges from "ROOT" to "r8". Each edge on the current task solution is also marked by the state space considered for planning. Green edges are ones for which motion plans have been found. Red edges are ones for which motion plans have not yet been found, but they are part of the currently considered task solution. Green vertices are task regions that can be reached with currently available motion plans. Blue vertices are task regions that could be planned towards from currently reached task regions. The labels for the edges in bold identify the state spaces that correspond to the TMM edges selected as the proposed sequence of actions.

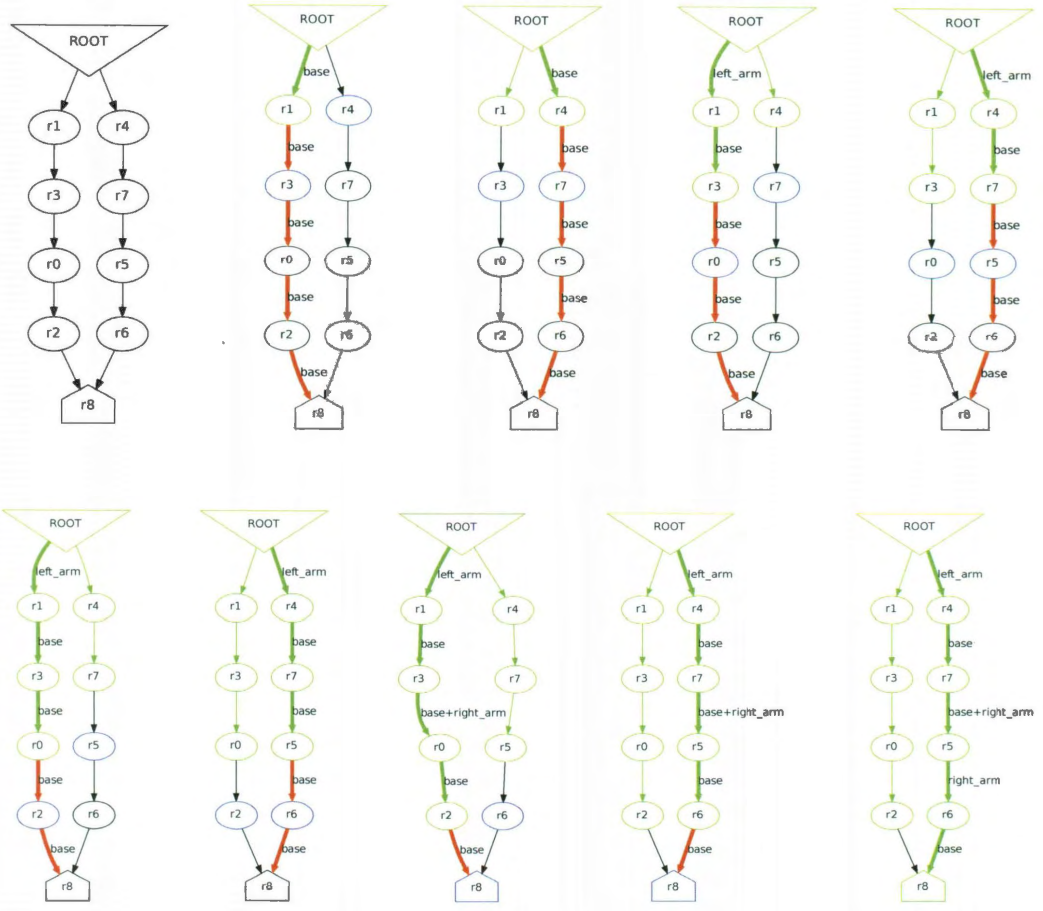



**Figure 5.5:** A sample step by step execution of Algorithm 5 (left to right and top to bottom) for the "Office1" environment (shown in Figure 5.6).

**Collected Experimental Data** Tables 5.1, 5.2, 5.3 and 5.4 show the results of solving the task and motion planning problem with the TMM2, TMM1 and graph approaches, for the environments described above, averaged over 30 runs. Each table shows the success rate of the approaches (the percentage of times a solution was produced within the allowed time), the amount of time spent planning motions, the amount of memory consumed by exploration data-structures, the length of the produced solution and the percentage of edges in the TMM (and graph, respectively) used while searching for a solution. The length of a TMM plan (see Definition 5.2.3) is the sum of the lengths of its motion plans, and the length of a motion plan is the sum of the distances between its way-points (as defined in Section 1.1.1):

$$\begin{aligned} d(x, x') \;=\; & d_2(Project(x, \mathcal{X}_{J_{base}}), Project(x', \mathcal{X}_{J_{base}})) \cdot 0.05 \;+ \\ & d_2(Project(x, \mathcal{X}_{J_{left\_arm}}), Project(x', \mathcal{X}_{J_{left\_arm}})) \;+ \\ & d_2(Project(x, \mathcal{X}_{J_{right\_arm}}), Project(x', \mathcal{X}_{J_{right\_arm}})), \end{aligned}$$

where $d_2$ stands for the L2 norm. The factor 0.05 was used for the base to compensate for the size of the environment. The angles of the joints in the arms were measured in radians.

For each of the reported measurements, the method that performed better is shown in bold face. The time spent planning motions and the length of the obtained solutions are also shown in Figure 5.7, Figure 5.9, Figure 5.11 and Figure 5.13.

For "Office1", on average, TMM2 seems to be the better approach (up to 20% faster than the graph approach), followed by TMM1 (in the range of 15% faster than

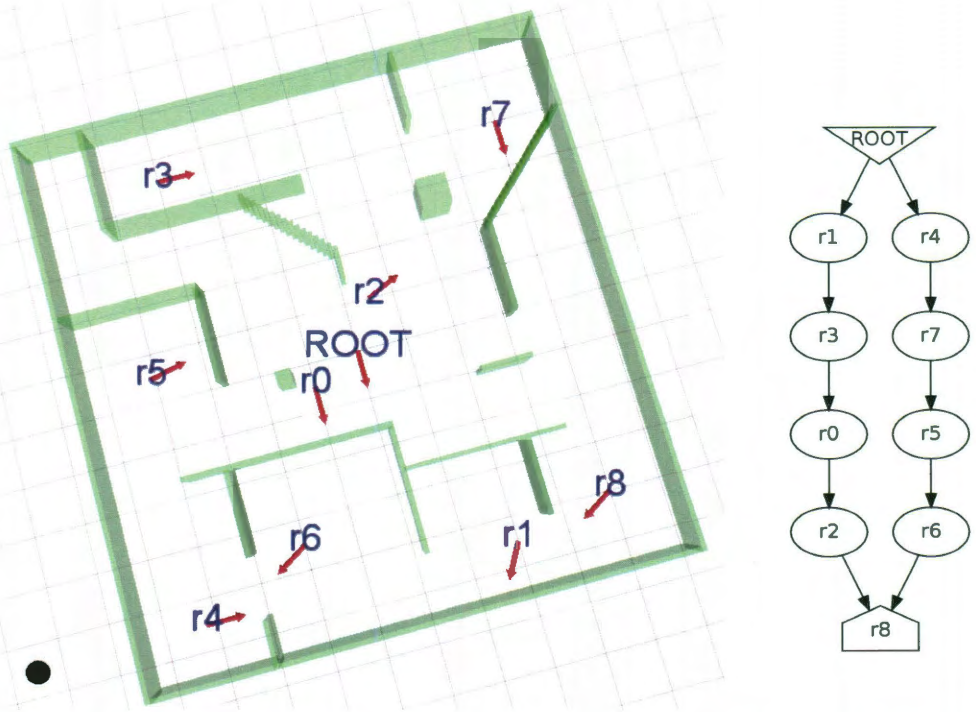



**Figure 5.6:** Left: The "Office1" environment. Right: The TMG for the task to solve.

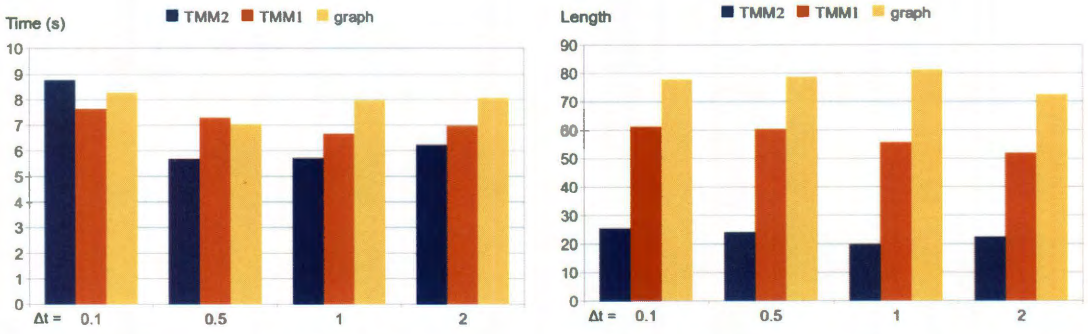



**Figure 5.7:** Planning time and solution length for the "Office1" problem (as in Table 5.1).

| $\Delta t$ (s) | | success | time (s) | mem (MB) | length | edges |
|---|---|---|---|---|---|---|
| | TMM2 | 100% | 8.77 | 1.00 | **25.70** | **49%** |
| 0.10 | TMM1 | 100% | **7.64** | **0.36** | 61.36 | 65% |
| | graph | 100% | 8.27 | 0.47 | 77.87 | 70% |
| | TMM2 | 100% | **5.69** | 0.60 | **24.17** | **34%** |
| 0.50 | TMM1 | 100% | 7.29 | **0.35** | 60.45 | 57% |
| | graph | 100% | 7.04 | 0.42 | 78.70 | 68% |
| | TMM2 | 100% | **5.73** | 0.48 | **19.92** | **29%** |
| 1.00 | TMM1 | 100% | 6.67 | **0.31** | 55.68 | 52% |
| | graph | 100% | 7.96 | 0.46 | 81.26 | 67% |
| | TMM2 | 100% | **6.24** | 0.54 | **22.52** | **28%** |
| 2.00 | TMM1 | 100% | 6.97 | **0.34** | 51.89 | 54% |
| | graph | 100% | 8.06 | 0.45 | 72.52 | 72% |

**Table 5.1:** Experimental results for the "Office1" problem (shown in Figure 5.6)

| $\Delta t$ (s) | | success | time (s) | mem (MB) | length | edges |
|---|---|---|---|---|---|---|
| | TMM2 | 100% | 1.03 | 0.19 | **43.61** | **35%** |
| 0.10 | TMM1 | 100% | 0.71 | 0.04 | 54.02 | 88% |
| | graph | 100% | **0.48** | **0.03** | 45.62 | 81% |
| | TMM2 | 100% | 1.03 | 0.22 | **43.96** | **32%** |
| 0.50 | TMM1 | 100% | 0.74 | 0.04 | 50.02 | 89% |
| | graph | 100% | **0.56** | 0.04 | 45.51 | 92% |
| | TMM2 | 100% | 1.12 | 0.23 | **44.56** | **32%** |
| 1.00 | TMM1 | 100% | 0.90 | 0.05 | 48.10 | 89% |
| | graph | 100% | **0.60** | **0.04** | 45.33 | 92% |
| | TMM2 | 100% | 1.26 | 0.26 | **44.47** | **32%** |
| 2.00 | TMM1 | 100% | 0.86 | 0.04 | 49.43 | 89% |
| | graph | 100% | **0.62** | 0.04 | 45.24 | 92% |

**Table 5.2:** Experimental results for the "Office2" problem (shown in Figure 5.8)

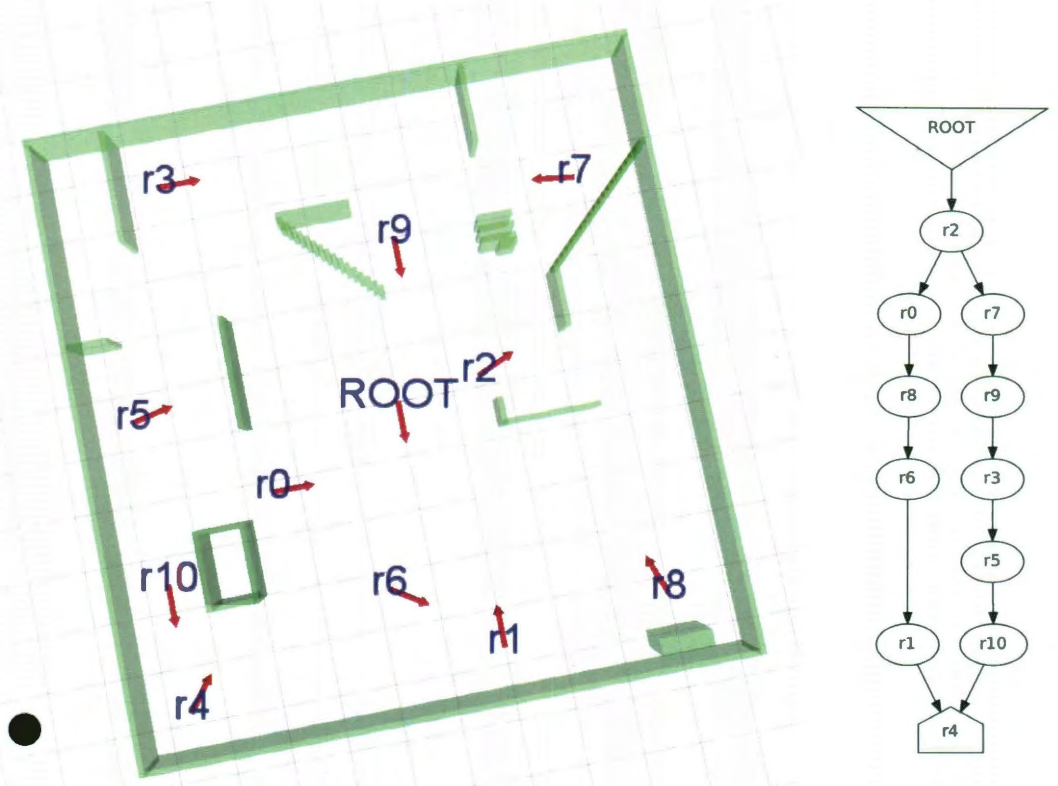



**Figure 5.8:** Left: The "Office2" environment. Right: The TMG for the task to solve.

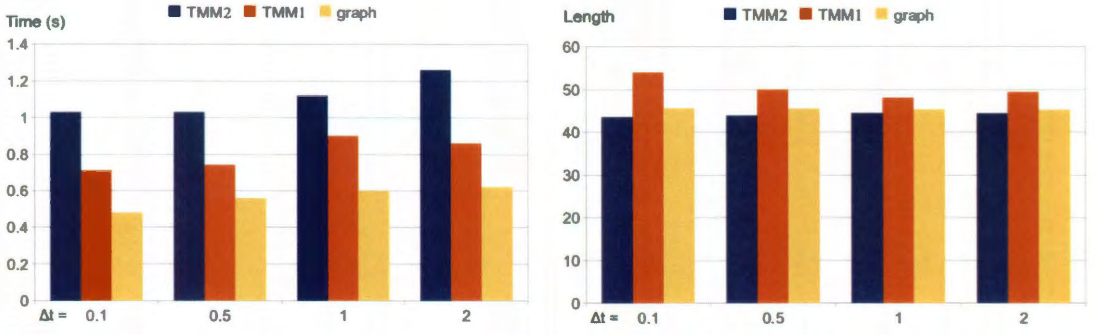



**Figure 5.9:** Planning time and solution length for the "Office2" problem (as in Table 5.2).


**Figure 5.10:** Left: The "Office3" environment. Right: The TMG for the task to solve.

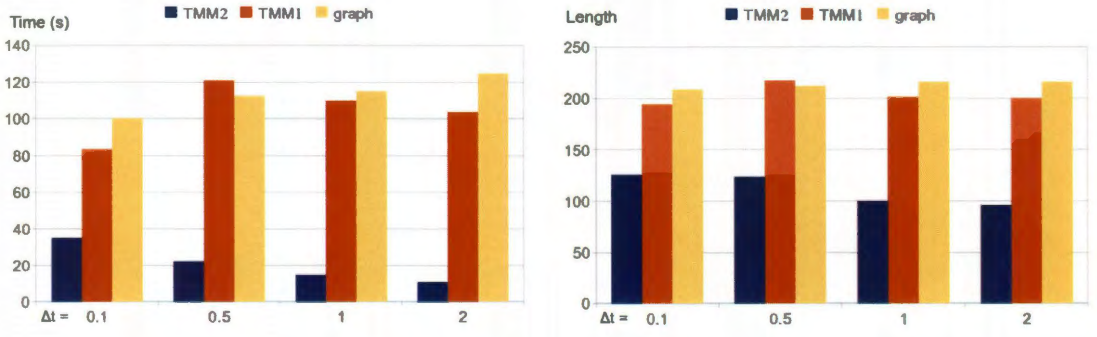



**Figure 5.11:** Planning time and solution length for the "Office3" problem (as in Table 5.3).

| $\Delta t$ (s) | | success | time (s) | mem (MB) | length | edges |
|---|---|---|---|---|---|---|
| 0.10 | TMM2 | **100%** | **35.40** | **3.33** | **126.34** | 81% |
| | TMM1 | 60% | 83.36 | 4.14 | 194.64 | 76% |
| | graph | 63% | 100.17 | 4.58 | 208.95 | **74%** |
| 0.50 | TMM2 | 100% | **22.34** | **2.35** | **124.18** | **68%** |
| | TMM1 | 100% | 120.91 | 6.29 | 217.47 | 85% |
| | graph | 100% | 112.53 | 5.70 | 212.17 | 86% |
| 1.00 | TMM2 | 100% | **15.04** | **1.71** | **100.50** | **47%** |
| | TMM1 | 100% | 109.93 | 5.83 | 201.68 | 84% |
| | graph | 100% | 115.22 | 5.78 | 216.30 | 88% |
| 2.00 | TMM2 | 100% | **11.09** | **1.47** | **96.50** | **34%** |
| | TMM1 | 100% | 103.85 | 5.54 | 200.31 | 82% |
| | graph | 100% | 124.80 | 6.36 | 216.30 | 86% |

**Table 5.3:** Experimental results for the "Office3" problem (shown in Figure 5.10)

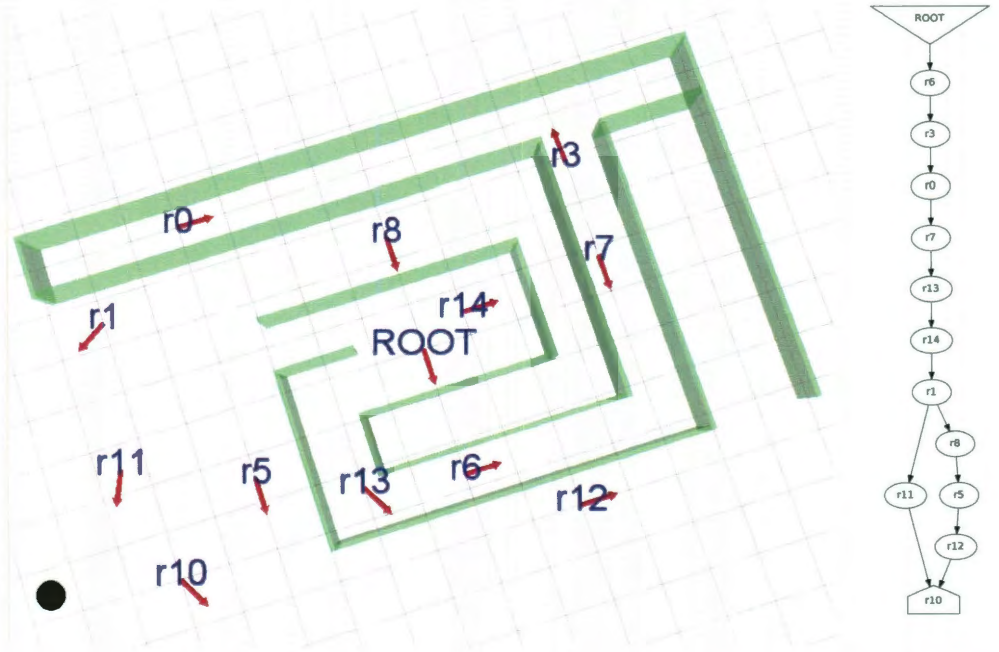



**Figure 5.12:** Left: The "Winding Tunnels" environment. Right: The TMG for the task to solve.

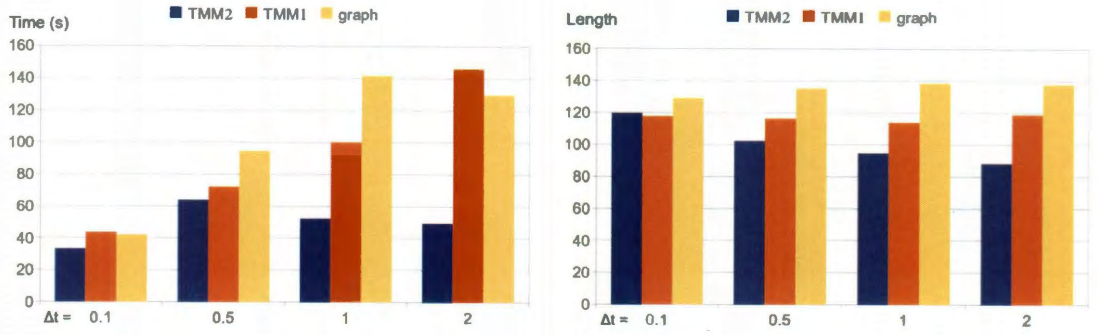



**Figure 5.13:** Planning time and solution length for the "Winding Tunnels" problem (as in Table 5.4).

| $\Delta t$ (s) | | success | time (s) | mem (MB) | length | edges |
|---|---|---|---|---|---|---|
| 0.10 | TMM2 | 40% | **33.31** | 3.04 | 119.96 | **53%** |
| | TMM1 | 40% | 43.55 | **2.13** | **117.72** | 62% |
| | graph | 20% | 41.94 | 2.54 | 129.25 | 54% |
| 0.50 | TMM2 | **100%** | **63.85** | 4.71 | **102.25** | **48%** |
| | TMM1 | 90% | 71.98 | **4.32** | 116.29 | 75% |
| | graph | 90% | 94.60 | 5.30 | 135.32 | 80% |
| 1.00 | TMM2 | 100% | **52.23** | **3.71** | **94.72** | **43%** |
| | TMM1 | 100% | 99.93 | 5.12 | 113.82 | 76% |
| | graph | 97% | 141.78 | 8.08 | 138.57 | 84% |
| 2.00 | TMM2 | 100% | **49.41** | **3.45** | **88.19** | **39%** |
| | TMM1 | 100% | 146.00 | 7.73 | 118.80 | 75% |
| | graph | 97% | 129.66 | 7.58 | 137.69 | 84% |

**Table 5.4:** Experimental results for the "Winding Tunnels" problem (shown in Figure 5.12)

the graph approach). For "Office2", the ranking of average runtimes of the methods is reversed: the graph approach is fastest and TMM2 is slowest. However, "Office2" is a very simple problem, and all runtimes are in the range of 1 second.

For simpler problems, such as "Office1" and "Office2", TMM2 and TMM1 do not provide a significant computational benefit in terms of runtime. However, shorter

solution paths (up to a factor of three in "Office1") are observed when using TMM2. TMM1 provides shorter solutions for "Office1" as well. The amount of time allowed for a planning step ($\Delta t$) does not significantly influence the trend of the results.

When computing solutions for more complex problems, such as "Office3" and "Winding Tunnels", a speedup factor of three to six is obtained with TMM2. At the same time, the solutions produced with TMM2 are shorter by as much as a factor of two. "Office3" is an artificial example designed specifically to expose the benefits of TMM2 over TMM1: planning for all the 17 degrees of freedom of the robot is almost always required, so the performance of TMM1 degrades towards that of the graph approach, both in terms of runtime and path length. Using TMM2, decoupled planning comes to play and significantly shorter solutions are obtained, in addition to the much reduced runtime.

For the "Winding Tunnels" problem, a situation similar to that for "Office1" is observed: TMM2 is the fastest method, but significant improvements are observed with TMM1 as well. Because "Winding Tunnels" is a more difficult problem, the improvements due to TMMs are more pronounced. The value of $\Delta t$ again does not influence the trend of the computation time and that of the solution length, but for low values of $\Delta t$, the success rate of all evaluated algorithms drops. This drop occurs because more time is spent updating the considered sequence of actions and less time is spent computing motion plans.

The percentage of edges that are used for motion planning is usually lower for TMMs rather than for graphs. This is to be expected because TMMs have a much larger number of edges. An additional observation is that when $\Delta t$ is smaller, a larger

percentage of edges is considered. This is again to be expected because with lower values of $\Delta t$, more sequences of actions are considered by the TMM algorithm.

## 5.6 Discussion and Possible Extensions

As described, the TMM edges always correspond to motion planning in a particular state space. This can be generalized to include other means of generating motions for TMM edges. For example, it could be possible to attempt simply controlling the robot along a straight path for a TMM edge, and only when that controller fails to find a solution, resort to motion planning. For complex manipulation scenarios, the use of a manipulation graph along certain TMM edges may be beneficial as well. Of course, in order to consider additional types of TMM edges, the cost functions need to be updated accordingly.

In all the described experiments, $2^3 - 1$ edges were included in the TMM for every action in the TMG. In general, this does not have to be the case. For example, to move a robot's head, it is unlikely that planning in the space that corresponds to the robot's arms will lead to a solution. As long as the full state space of the robot is considered, not all the possible combinations of subspaces have to be included in the TMM to maintain the completeness guarantees of the underlying motion planner.

Another possible extension is to use a lazy form of propagating information between state spaces in Algorithm 6. Such an approach would reduce and even eliminate any overhead caused by the process of sharing information described above.

# Chapter 6

# Uncertainty and Task Motion Multigraphs

Physical systems are characterized by imperfect actuation, imperfect sensing and imperfect models, all of which lead to uncertainty. The sequence of actions a robot follows to reach its goal influences the quality of the sensed data. For example, a robot navigating in an empty environment will have more difficulty in localization when compared to a robot that navigates around corners and walls. At the same time, different actuators have different abilities in terms of following planned paths, and the set of actuators used depends on the state spaces in which motions were planned. As such, the information TMMs include, i.e., the possible sequences of actions that lead to the goal and the state spaces that could be potentially used to plan motions for those actions, directly affects a robot's capability to address uncertainty issues.

This chapter shows how to incorporate uncertainty information at the task plan-

ning level, using TMMs. A user can easily specify uncertainty information for edges and vertices of the TMM (e.g., a notion of safety that is specific to particular vertices of particular edges). Furthermore, this chapter assumes that motion planners capable of reporting probabilities of success for their computed solutions are used as underlying motion planners in the TMM-based algorithms described in Chapter 5. Section 6.1 presents some of the previous work in considering uncertainty in motion planning, and includes examples of motion planners that are able to qualify their reported solutions with probabilities of success. However, for simplicity of implementation, this work does not actually use planners such as the ones described in Section 6.1. Instead, an artificial model of uncertainty is employed to emulate uncertainty in localization. This is a simple source of uncertainty and it is meant solely as an example (many other different sources of uncertainty can be used). Section 6.2 shows how to incorporate information about uncertainty at the task planning level using MDPs. Experimental results are shown in Section 6.3.

## 6.1 Considering Uncertainty in Motion Planning

Significant progress has been made towards incorporating various forms of uncertainty at the motion planning level, leading to algorithms that compute robust motion plans. These algorithms can be separated in two categories: ones that plan in the state space and ones that plan in the belief space.

### 6.1.1 Planning in the State Space

This class of algorithms builds upon already proven sampling-based algorithms such as PRM [36], and estimate probabilities of success for segments that make up solution paths. Techniques that plan in the state space can consider different types of uncertainty, but do not necessarily consider all types of uncertainty simultaneously. For example, uncertainty in sensing (e.g. [111, 112]) and uncertainty in actuation (e.g. [113, 114]) can be considered separately. Nevertheless, considering both uncertainty in sensing and uncertainty in actuation is possible [115].

Missiuro and Roy present a modified version of PRM that can account for uncertainty in the representation of the environment [111]. It is assumed that the the vertices that make up the sensed obstacles in the environment are represented using Gaussian probability distributions, and uncertainty in the robot state is ignored. The roadmap construction step of PRM is modified so that the probability of collision for sampled states is estimated. A sampled state is then discarded with probability equal to its probability of collision. At the query step of PRM, the probability of collision for a segment in the roadmap is evaluated using a Monte Carlo approach. The cost of a segment is then defined as a weighted sum that considers the probability of collision and the length of the segment. The reported solution is the shortest path connecting the given start and goal states.

Alterovitz et al. present another modification of PRM which they call the Stochastic Roadmap Method (SMR) [113]. Their approach considers uncertainty in actuation and is applied for of a system that accepts a finite set of controls. A stochastic model of motion is assumed, and probabilities for successful execution are computed for each

edge in the roadmap. A Markov Decision Process [116] is used in the query step of PRM.

### 6.1.2 Planning in the Belief Space

Instead of planning in the state space of the robotic system, another approach is to plan in the belief space. A point in the belief space consists of the parameters necessary to represent a probability distribution for a point in the state space. For example, for a finite state space with $n$ states, the dimensionality of the belief space is equal to $n - 1$. For infinite state spaces, the dimension of the belief space depends on the probability model used. In previous work, Gaussian models of probability or sets of particles are often used to approximate the belief state.

An approach similar in implementation to planning in the state space is the belief roadmap [117], which can consider uncertainty in state information. The idea behind this approach is to run PRM in the belief space. Points in the belief space are Gaussian probability distributions. The dimensionality of the belief space is then $\Theta(n^2)$ for a state space of dimension $n$. Even though the robot is assumed to be fully actuated, when milestones sampled in the belief space are to be connected, only $n$ of the parameters can be controlled: the means of the belief states. To address this problem, Prentice and Roy [117] only sample the means of the belief states – which are in fact elements of the state space – and factorize the covariance matrices in a manner that affords computational savings. Although this method does reason in the belief space of the robotic system, it still does so by sampling in the state space.

Different approaches for formalizing uncertainty are based on Markov Decision

Processes [116] and Partially Observable Markov Decision Processes [118]. Markov Decision Processes (MDPs) can be used to model robots that have uncertainty in actuation, but their state is fully observable (known at all times), and Partially Observable Markov Decision Processes can be used to model robots that have uncertainty in actuation and their state is not fully observable. Because Markov Decision Processes (MDPs) are used in the following sections, they are defined next:

A Markov Decision Process (MDP) is a 4-tuple $(S, A, T, R)$, where:

- $S$ is the set of states the robot could be in.
- $A$ is the set of actions the robot can perform in order to move between states.
- $T : S \times A \to 2^S$ is a transition function indicating the probabilities of transition between states; $T(s, a, s')$ is the probability of transitioning from state $s$ to state $s'$ under action $a$. Time is discretized, and at every step the robot takes, the transition function is evaluated to determine which state the robot reaches.
- $R : S \times A \to \mathbb{R}$ is a reward function; $R(s', a)$ is the reward the robot receives if it reaches state $s'$ after performing action $a$. This reward is usually discounted over time using a parameter $\gamma$.

Often, the sets $S$ and $A$ are assumed to be finite. The solution to an MDP is a policy $\pi : S \to A$ which specifies what the optimal action is for every state the robot could

be at. The notion of optimality is defined in terms of maximizing rewards:

$$\begin{aligned} \pi(s) &= \arg\max_a \sum_{s'} T(s,a,s')(R(s',a) + \gamma V(s')) \\ V(s) &= \sum_{s'} T(s,\pi(s),s')(R(s',\pi(s)) + \gamma V(s')), \end{aligned}$$

where $\gamma$ is the discount factor for the MDP. This factor has the effect of diminishing the value of rewards the further they are in the future.

Typical algorithms that find the optimal policy for an MDP are policy iteration and value iteration [116].

## 6.2 Considering Uncertainty in Task and Motion Planning using TMMs

From a high-level perspective, the operation of Algorithm 5 can be viewed as interleaving *(1)* the proposal of a possible sequence of actions that take the robot to the goal and *(2)* the computation of some of the motion plans for the considered actions. If some form of uncertainty is considered at the motion planning level, it is possible to assign a probability of success to motion plans computed for the edges in the TMM. This section shows how to make use of such probabilities so that more robust sequences of actions are proposed in Algorithm 5. The step that proposes the sequence of actions to follow in Algorithm 5 is based on an assignment of costs to edges and Dijkstra's algorithm. In this section, the use of Dijkstra's algorithm is replaced by the use of Markov Decision Processes (MDPs) and value iteration.

Section 6.2.1 shows how to construct an MDP from a TMM at a given point in time and Section 6.2.2 shows how to use an MDP in Algorithm 5. Experimental results for the updated algorithm are shown in Section 6.3.

### 6.2.1 Construction of an MDP

Given the exploration information from a TMM $G_M = (V_M, E_M)$ at a particular time, an MDP $(S, A, T, R)$ is constructed as follows:

- $S = V_M \cup \{Fail\}$; the vertices in the TMM become states in the MDP, and an additional state $(Fail)$ that corresponds to catastrophic failure is added.
- $A = \bigcup_{e \in E_M} Space(e)$; all the state spaces that could be used for planning are considered actions in the MDP.
- The MDP state transition function $T$ is defined as follows:

$$
\begin{aligned}
T(s', s, Space(e)) &= \eta \cdot Pt(e), \text{ for each edge } e = (s, s') \in E_M \\
T(Fail, s, Space(e)) &= \eta \cdot \sum_{s' \in \{s' | (s,s') \in E_M\}} 1 - T(s', s, Space(e)) \\
Pt(e) &= pModel(e) \cdot \begin{cases} p & \text{if } sol \\ \frac{1}{2} \cdot \frac{1}{1+t} & \text{if not } sol, \end{cases}
\end{aligned}
$$

where $pModel(e)$ corresponds to model uncertainty (in this work it depends on the state space used to plan motions along $e$, but this definition could be extended), $p$ corresponds to the probability of success of a particular motion

plan (reported by a motion planner when computing that plan), $t$ is the amount of time spent planning motions for edge $e$, $\eta$ is a normalization constant such that the probabilities of outgoing transitions from a particular state under a particular action sum up to 1 (property of MDPs), and *sol* is a flag indicating whether any motion plans have been found for edge $e$.

The transition function brings information obtained at the motion planning level to the task planning level and thus allows the TMM-based algorithms discussed in Chapter 5 to make use of it. Intuitively, for every edge in the TMM, there exists a corresponding transition in the MDP. The probability of that transition depends on the state space of the system (given by $pModel(e)$) and the computed motion plan (given by $p$) when such a plan is found. If motion plans do not yet exist for the edge $e$, a probability of success that diminishes as computation time increases is assumed.

- The reward function $R$ of the MDP is defined such that:

$$
\begin{aligned}
R(Fail, \cdot) &= -10000 \\
R(s,a) &= \begin{cases} R^0(s,a) \cdot \begin{cases} dim(\mathcal{X}_J) - dim(Space(e)) & \text{if } p > P_C \\ dim(Space(e)) & \text{otherwise} \end{cases} & \text{if } sol \\ -\xi \cdot dim(Space(e)) & \text{if not } sol \end{cases} \\
R^0(s,a) &= \beta \cdot \left( \frac{1}{1 + e^{-\alpha \cdot (p - P_C)}} - \frac{1}{2} \right),
\end{aligned}
$$

where $\mathcal{X}_J$ is the full state space of the robotic system, $\alpha, \beta, \xi$ are scaling factors, $p$ is the probability of success of the most probable motion in $M(e)$, *sol* is a flag that indicates whether any motion plan has been found for an edge in $M(e)$ and $P_C$ is a trust threshold. Explanations for these variables follow.

The intention of the reward function is to penalize actions that are not desirable and to reward ones that are. At first, no probabilities are known for any of the actions in the TMM. In this case, the only reasonable approach is to consider a small penalty for each action in the corresponding MDP, so that we avoid a bias towards long policies.

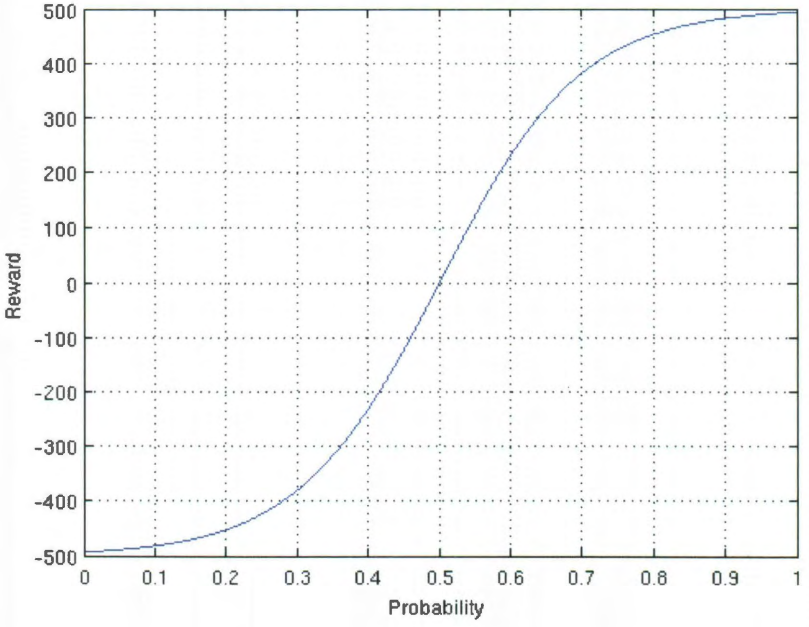



**Figure 6.1:** Value of $R^0(\cdot,\cdot)$ function used in defining rewards for $P_C = 0.5$, $\xi = 0.05, \alpha = 10$ and $\beta = 1000$.

When a motion plan is found for a particular edge in the TMM, the probability of that motion plan can either be sufficiently high so that including the corresponding MDP transition in the optimal policy is desirable, or it is a low probability and the corresponding MDP transition should be ignored. The distinction is made using the trust threshold $P_C$: edges that have $p > P_C$ will yield a reward that is positive, while ones that have $p < P_C$ will yield a negative reward; $\alpha$ and $\beta$ are positive factors that set the sale for the returned rewards. A representation of the $R^0(\cdot,\cdot)$ function is shown in Figure 6.1. $P_C$ is a key parameter, as it decides whether the reward is positive or not. When a probability $p$ is not available, because a solution has not yet been found, the reward achieved by the edge is considered to be a small negative value, and the $\xi$ factor (positive) is used.

In this thesis, the discount factor $\gamma$ for the constructed MDP is always set to 0.95. The constants for the definition of the reward functions are $P_C = 0.5, \xi = 0.05, \alpha = 10$ and $\beta = 1000$. Furthermore, $pModel(e)$ is defined as:

$$pModel(e) = \begin{cases} 0.99 & \text{if } Space(e) \in \{\mathcal{X}_{right\_arm}, \mathcal{X}_{left\_arm}\} \\ 0.90 & \text{if } Space(e) = \mathcal{X}_{base} \\ 0.75 & \text{otherwise.} \end{cases}$$

The values mentioned here are by no means fixed. For practical applications, these values will most likely need to be tuned. The intention in this thesis only to show that TMMs can easily be used to consider uncertainty at task level, and the values

selected are ones that lead to results supporting this point.

### 6.2.2 Using MDPs in the TMM Algorithm

Given an MDP constructed as shown above, value iteration is executed to find the optimal policy. Let $\pi : S \rightarrow A$ be the optimal policy. The sequence of states $s_0, s_1, \ldots, s_k$ is extracted from the MDP such that $s_0$ is the MDP state that corresponds to the *root* of the TMM,

$$s_{i+1} = \max_{s' \in S} \left( T(s', s_i, \pi(s_i)) \cdot R(s', \pi(s_i)) \right),$$

and $s_k$ corresponds to a goal state in the TMM. The sequence of states above is used to produce a sequence of possible actions that take the robot from the root of the TMM to one of the goals. This computation replaces the call to Dijkstra's algorithm in Algorithm 5 [line 2].

## 6.3 Experimental Results

**Source of Uncertainty** For the experiments shown in this section a simple model of uncertainty in localization was assumed: the farther the robot is from a wall in the environment, the higher the uncertainty in localization is. The level of uncertainty is shown in Figures 6.2, 6.3 and 6.4. The darker areas indicate locations that are farther from walls, and thus offer less localization information for the robot.

The values computed for this source of uncertainty are not realistic. Their purpose

is only to emphasize the fact that using information about probabilities at the task planning level improves the robustness of the final solution. For practical applications, different sources of uncertainty would be used.

**Experimental Setup** Algorithm 5 is updated as indicated in Section 6.2 and executed on the environments shown in Figures 6.2, 6.3 and 6.4. The environment in 6.2 is very small and very simple; its purpose is to show a motivating example for considering uncertainty at the task level as well. The environment in Figure 6.3 is fairly cluttered, and good localization information is available throughout the environment. The environment in Figure 6.4 however, has a large open area that does not provide good localization information. The environments were selected as such to emphasize the difference made by the use of MDPs.

All experiments in this chapter were conducted in the same fashion as the ones in Chapter 5. All values are averaged over 30 runs, and the motion planner used is RRT-Connect [41] from OMPL [93]. In addition to information reported in Chapter 5, this chapter includes two additional measurements: *(1)* "pct" represents the amount of time spent in the computation of proposed sequences of actions that connect the TMM's root to one of its goals, and *(2)* "prob" represents the probability of success for the complete task plan.

**Presentation of Results** Figure 6.2 shows a very simple problem where the robot has two options: move directly to the goal (from "ROOT" to "r2") in one single action, or take two additional actions, so that dark (unsafe) areas are avoided. In the process of the computation using Dijkstra's algorithm, the shortest path is the

only one considered, the motion plan between "ROOT" and "r2" is trivial to find, and the problem is solved. When using MDPs, the direct solution is computed at first. However, the probability of success for the motion plan connecting "ROOT" to "r2" is computed to be 0.1, which is less than $P_C$. This makes the reward of moving to "r2" directly very low, so the subsequently proposed sequences of actions extracted using MDPs are via regions "r0" and "r1". The probabilities of success for the corresponding motion plans are higher. Table 6.1 shows the values of the collected measurements. Rows indicated by "Dij." correspond to the use of Dijkstra's algorithm in Algorithm 5 and rows indicated by "MDP" correspond to the use of MDPs, as explained in Section 6.2. The probability of success when using MDPs is much larger than when using Dijkstra's algorithm; however, the number of edges in the TMM that are used in planning is significantly higher when using MDPs. Furthermore, the execution of Dijkstra's algorithm is much faster than value iteration (the "pct" column), and the memory consumption is increased when using MDPs. All these results are in some sense expected, just as is the fact that the length of the computed task solutions is higher for MDPs. The differences are small in this particular experiment because of the small scale of the environment and the used distance function.

An additional experiment that was conducted on the environment in Figure 6.2 was that the probability of success for the computed motion plans were always set to 1. In that case, the solution proposed using the MDP algorithm was always the same as the one using Dijkstra's algorithm. This is desirable, because shorter solutions are preferred when safety is not an issue.

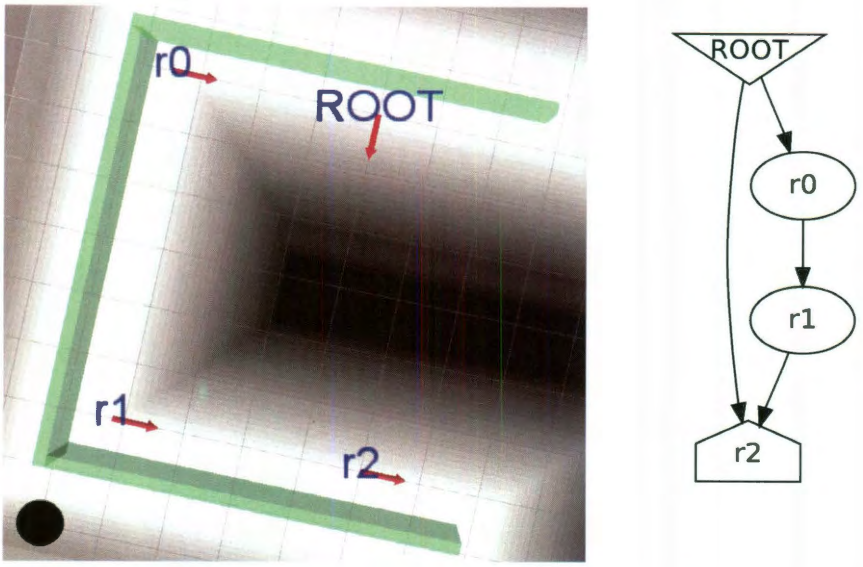



**Figure 6.2:** Motivating example for considering uncertainty. The obvious shorter path is to move from "ROOT" to "r2", crossing the dark (unsafe) area. Considering uncertainty produces the longer (but safer) solution, via the "r0" and "r1" regions.

| $\Delta t$ (s) | | success | time (s) | mem (MB) | pct (ms) | length | edges | prob |
|---|---|---|---|---|---|---|---|---|
| 0.10 | Dij. | 100% | **0.09** | **0.02** | **0.1** | **11.37** | **12**% | 0.09 |
| | MDP | 100% | 0.16 | 0.05 | 23.6 | 12.13 | 36% | **0.61** |
| 0.50 | Dij. | 100% | **0.09** | **0.02** | **0.1** | **11.43** | **11**% | 0.09 |
| | MDP | 100% | 0.16 | 0.05 | 22.8 | 12.15 | 36% | **0.61** |
| 1.00 | Dij. | 100% | **0.07** | **0.02** | **0.1** | **11.35** | **11**% | 0.09 |
| | MDP | 100% | 0.15 | 0.05 | 17.1 | 12.14 | 36% | **0.61** |
| 2.00 | Dij. | 100% | **0.07** | **0.02** | **0.1** | **11.38** | **11**% | 0.09 |
| | MDP | 100% | 0.18 | 0.05 | 22.6 | 12.19 | 36% | **0.61** |

**Table 6.1:** Experimental results for the motivating example shown in Figure 6.2

"Office1" shows an environment that includes few dark areas. As a result, the probability of success is only slightly higher when using MDPs rather than Dijkstra's algorithm, as shown in Table 6.2. An interesting side-effect of using MDPs however is that even though value iteration is much slower than execution of Dijkstra's algorithm ("pct" column), that difference is surpassed by the significantly reduced computation time when using MDPs. Because the structure of the "Office1" problem is such that there are two parallel sets of actions that lead to the goal, the overall costs of the two task paths alternate, making the version of the TMM algorithm that uses Dijkstra's algorithm compute motion plans for both task paths. When the version of the algorithm using MDPs is employed, the reward of one path becomes larger as motions deemed "safe" are found. As a result, task paths do not alternate and the algorithm stays committed to one path, as long as unsafe areas are not encountered. A side-effect of this behavior is reduced computation time, for this particular problem.

"Office2" is a version of "Office1" that includes darker areas too. The observations are similar as for "Office1", as shown in Table 6.3. The main difference is that the probability of success when using MDPs is distinctly higher, as the algorithm avoids the darker areas and follows the longer task path. The differences in runtime are also reduced. However, the actual value for the probability of success is still low even for the experiments using MDPs. The reason for these low values is the unrealistic model of uncertainty that was used and the fact that in this work, sampling-based planners ignored the uncertainty information (although, previous work has shown different possibilities of incorporating this information at the motion planning level).

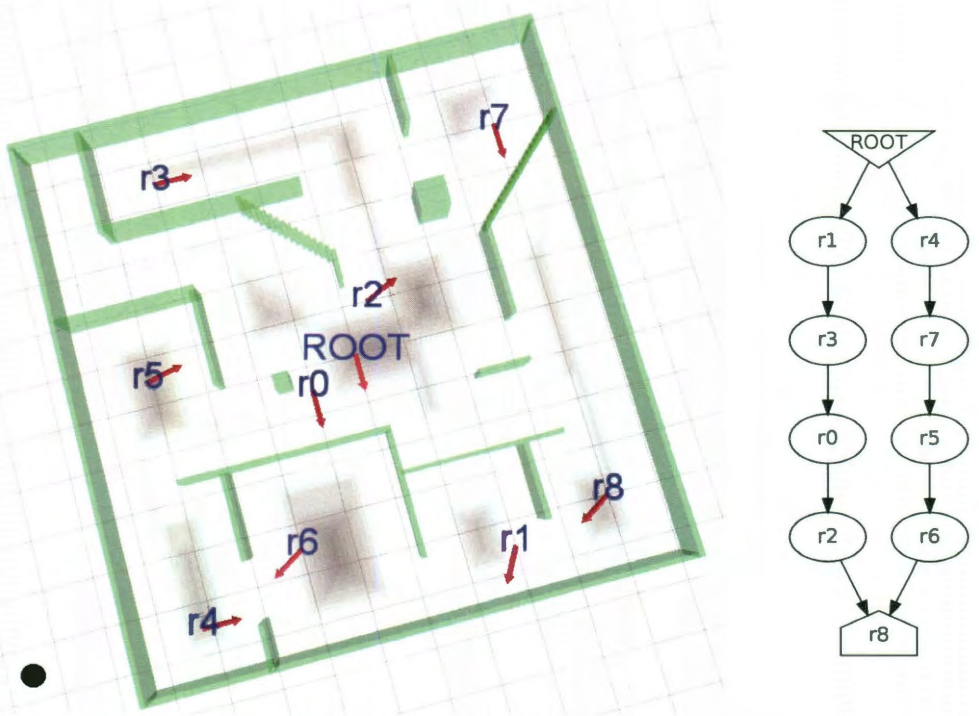



**Figure 6.3:** Left: The "Office1" environment with uncertainty map for localization. Darker areas cause poor localization. Right: The TMG for the task to solve.

| $\Delta t$ (s) | | success | time (s) | mem (MB) | pct (ms) | length | edges | prob |
|---|---|---|---|---|---|---|---|---|
| 0.10 | Dij. | 100% | **8.60** | **1.00** | **2.7** | **27.07** | **48%** | **0.45** |
| | MDP | 100% | 12.83 | 1.40 | 816.1 | 42.52 | 54% | 0.44 |
| 0.50 | Dij. | 100% | 6.28 | 0.64 | **0.7** | **23.20** | 35% | 0.45 |
| | MDP | 100% | **4.77** | **0.53** | 154.8 | 32.21 | **32%** | 0.45 |
| 1.00 | Dij. | 100% | 5.19 | 0.47 | **0.6** | **19.14** | 28% | 0.46 |
| | MDP | 100% | **1.91** | **0.25** | 102.7 | 23.96 | **21%** | **0.47** |
| 2.00 | Dij. | 100% | 6.91 | 0.60 | **0.5** | **19.20** | 29% | 0.44 |
| | MDP | 100% | **1.90** | **0.25** | 102.8 | 24.18 | **19%** | **0.46** |

**Table 6.2:** Experimental results for the "Office1" problem (shown in Figure 6.3)

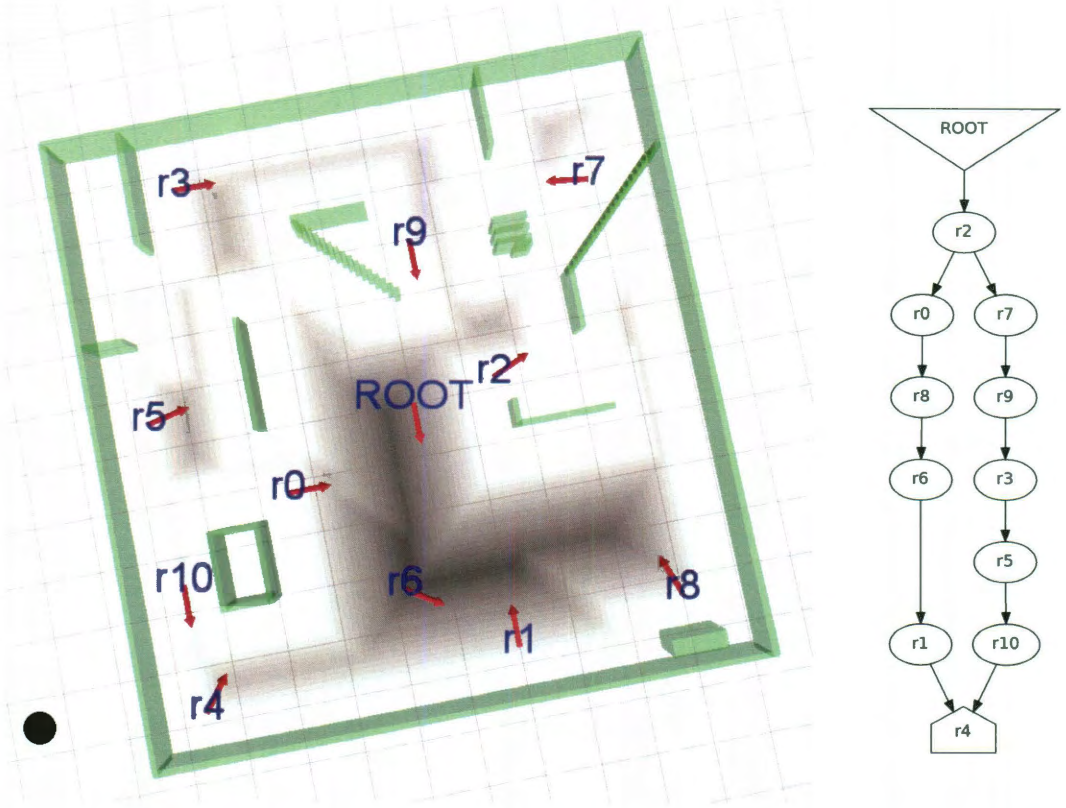



**Figure 6.4:** Left: The "Office2" environment with uncertainty map for localization. Darker areas cause poor localization. Right: The TMG for the task to solve.

| $\Delta t$ (s) | | success | time (s) | mem (MB) | pct (ms) | length | edges | prob |
|---|---|---|---|---|---|---|---|---|
| 0.10 | Dij. | 100% | **1.04** | **0.19** | **0.5** | **43.43** | **32%** | 0.33 |
| | MDP | 100% | 2.05 | 0.32 | 332.9 | 55.34 | 53% | **0.48** |
| 0.50 | Dij. | 100% | 1.12 | 0.23 | **0.5** | **44.37** | 32% | 0.34 |
| | MDP | 100% | **1.04** | **0.20** | 179.6 | 50.32 | **27%** | **0.52** |
| 1.00 | Dij. | 100% | 1.16 | 0.25 | **0.7** | **44.32** | 32% | 0.34 |
| | MDP | 100% | **0.99** | **0.20** | 172.3 | 50.26 | **26%** | **0.53** |
| 2.00 | Dij. | 100% | 1.21 | 0.26 | **0.7** | **44.18** | 32% | 0.34 |
| | MDP | 100% | **1.02** | **0.20** | 180.3 | 50.00 | **26%** | **0.52** |

**Table 6.3:** Experimental results for the "Office2" problem (shown in Figure 6.4)

## 6.4 Discussion and Possible Extensions

The use of MDPs as shown in this chapter is a simple approach of considering uncertainty at the task level. As shown by the conduced experiments, the robustness of the computed solutions can be significantly increased. Extensions to this method that allow consideration of cycles in TMMs could make the approach more general. Furthermore, different methods of constructing MDPs from TMMs could be evaluated.

# Chapter 7

# Conclusions

This thesis shows and experimentally validates algorithmic improvements for the *motion planning* problem and for the *simultaneous task and motion planning* (STAMP) problem.

An improved version of the KPIECE algorithm is developed. State space coverage estimation techniques based on projections to lower-dimensional Euclidean spaces lead to significantly more efficient sampling-based planners. The described improvements, as well as a number of previously introduced algorithms, are included in a free software library called OMPL. The practical applicability of KPIECE and OMPL is demonstrated on the PR2 mobile manipulator.

The notion of a task motion multigraph (TMM) is introduced to help with solving the STAMP problem. This thesis shows that the exchange of information between selection of tasks and computation of motion plans leads to increased efficiency in planning. In the case of planning under geometric constraints, speedups by as much

as a factor of six are observed. Furthermore, solution paths that are shorter and require the actuation of fewer hardware components are shown for the PR2 mobile manipulator. If uncertainty information is available at the motion planning level, TMMs facilitate the inclusion of that information in the computation of task plans through the use of Markov Decision Processes.

The individual improvements this thesis brings are applicable to a variety of problems, but the thesis as a whole furthers the planning capabilities of mobile manipulators.

# Bibliography


[1] J.-C. Latombe, *Robot Motion Planning.* Boston, MA: Kluwer Academic Publishers, 1991.

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 2003.

[3] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations.* MIT Press, June 2005.

[4] S. M. LaValle, *Planning Algorithms.* Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.

[5] J.-C. Latombe, "Motion planning: A journey of robots, molecules, digital actors, and other artifacts," *International Journal of Robotics Research*, vol. 18, no. 11, pp. 1119–1128, 1999.

[6] S. Thomas, X. Tang, L. Tapia, and N. M. Amato, "Simulating protein motions with rigidity analysis." *Journal of Computational Biology*, vol. 14, no. 6, pp. 839–855, 2007.

[7] M. S. Apaydin, D. L. Brutlag, C. Guestrin, D. Hsu, J.-C. Latombe, and C. Varma, "Stochastic roadmap simulation: an efficient representation and algorithm for analyzing molecular motion." *Journal of Computational Biology*, vol. 10, no. 3-4, pp. 257–281, 2003.

[8] E. Plaku, L. E. Kavraki, and M. Y. Vardi, "Hybrid systems: from verification to falsification by combining motion planning and discrete search," Ph.D. dissertation, Rice University, 2009.

[9] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation planning with probabilistic roadmaps," *International Journal of Robotics Research*, vol. 23, pp. 729–746, 2004.

[10] K. Hauser and J.-C. Latombe, "Integrating task and PRM motion plannning," in *International Conference on Automated Planning and Scheduling*, 2009, workshop on Bridging the Gap between Task and Motion Planning.

[11] J. Guitton and J.-L. Farges, "Taking into account geometric constraints for task-oriented motion planning," in *International Conference on Automated Planning and Scheduling*, 2009, workshop on Bridging the Gap between Task and Motion Planning.

[12] S. Cambon, R. Alami, and F. Gavrot, "A hybrid approach to intricate motion, manipulation and task planning," *International Journal of Robotics Research*, vol. 28, pp. 104–126, 2009.

[13] J. Choi and E. Amir, "Combining planning and motion planning," in *IEEE International Conference on Robotics and Automation*, Kobe, Japan, May 2009, pp. 238–244.

[14] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation," in *International Conference on Automated Planning and Scheduling*, 2010, pp. 254–258.

[15] L. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *IEEE International Conference on Robotics and Automation.* Anchorage, Alaska: Workshop on Mobile Manipulation, May 2010.

[16] "Open dynamics engine." [Online]. Available: http://www.opende.sourceforge.net

[17] J. Boren and S. Cousins, "The SMACH high-level executive," in *IEEE Robotics & Automation Magazine*, December 2010, vol. 17, no. 4, pp. 18–20.

[18] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, 2000.

[19] D. McDermott, "The 1998 AI planning systems competition," *AI Magazine*, vol. 21, no. 2, pp. 35–55, 2000.

[20] G. Sánchez and J.-C. Latombe, "A single-query bi-directional probabilistic roadmap planner with lazy collision checking," *International Journal of Robotics Research*, vol. 6, pp. 403–417, 2003.

[21] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *IEEE International Conference on Robotics and Automation*, vol. 3, Albuquerque, New Mexico, April 1997, pp. 2719–2726.

[22] ROS: Robot Operating System. [Online]. Available: http://www.ros.org

[23] J. T. Schwartz and M. Sharir, "On the piano movers' problem: General techniques for computing topological properties of real algebraic manifolds," *Communications on Pure and Applied Mathematics*, vol. 36, pp. 345–398, 1983.

[24] ——, "On the piano movers' problem: III. coordinating the motion of several independent bodies: The special case of circular bodies moving amidst polygonal barriers," *International Journal of Robotics Research*, vol. 2, no. 3, pp. 46–75, 1983.

[25] M. Sharir and E. Ariel-Sheffi, "On the piano movers' problem: IV. various decomposable two-dimensional motion-planning problems," *Communications on Pure and Applied Mathematics*, vol. 37, no. 4, pp. 479–493, 1983.

[26] J. H. Reif, "Complexity of the mover's problem and generalizations," in *IEEE Symposium on Foundations of Computer Science*, 1979, pp. 421–427.

[27] J. Canny, "Some algebraic and geometric computations in PSPACE," in *Annual ACM Symposium on Theory of Computing.* Chicago, Illinois, United States: ACM Press, 1988, pp. 460–469.

[28] J. Barraquand and J.-C. Latombe, "Robot motion planning : A distributed representation approach," *International Journal of Robotics Research*, vol. 10, no. 6, pp. 628–649, 1991.

[29] P. Cheng, G. Pappas, and V. Kumar, "Decidability of motion planning with differential constraints," in *IEEE International Conference on Robotics and Automation*, Rome, Italy, April 2007, pp. 1826–1831.

[30] B. Donald, P. Xavier, J. Canny, and J. Reif, "Kinodynamic motion planning," *Journal of the ACM*, vol. 40, no. 5, pp. 1048–1066, 1993.

[31] J. H. Reif and S. Slee, "Optimal kinodynamic motion planning for self-reconfigurable robots between arbitrary 2d configurations," in *Robotics: Science and Systems*, W. Burgard, O. Brock, and C. Stachniss, Eds. Atlanta, Georgia: MIT Press, June 2007.

[32] I. A. Şucan, J. F. Kruse, M. Yim, and L. E. Kavraki, "Kinodynamic motion planning with hardware demonstrations," in *International Conference on Intelligent Robots and Systems*, Nice, France, September 2008, pp. 1661–1666.

[33] R. B. Rusu, I. A. Şucan, B. Gerkey, S. Chitta, M. Beetz, and L. E. Kavraki, "Real-time perception guided motion planning for a personal robot," in *International Conference on Intelligent Robots and Systems*, St. Louis, Missouri, October 2009, pp. 4245–4252.

[34] L. E. Kavraki, J.-C. Latombe, R. Motwani, and P. Raghavan, "Randomized query processing in robot path planning," *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 50–60, 1998.

[35] A. Ladd and L. Kavraki, "Measure theoretic analysis of probabilistic path planning," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 2, pp. 229–242, 2004.

[36] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, August 1996.

[37] J. Barraquand, L. E. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, and P. Raghavan, "A random sampling scheme for robot path planning," *International Journal of Robotics Research*, vol. 16, no. 6, pp. 759–774, 1997.

[38] L. Kavraki, M. Kolountzakis, and J.-C. Latombe, "Analysis of probabilistic roadmaps for path planning," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 1, pp. 166–171, Feb. 1998.

[39] C. Nissoux, T. Simeon, and J.-P. Laumond, "Visibility based probabilistic roadmaps," in *International Conference on Intelligent Robots and Systems*, vol. 3, Kyongju, Korea, October 1999, pp. 1316–1321.

[40] R. Bohlin and L. Kavraki, "Path planning using Lazy PRM," in *IEEE International Conference on Robotics and Automation*, San Francisco, California, April 2000, pp. 521–528.

[41] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *IEEE International Conference on Robotics and Automation*, San Francisco, California, April 2000, pp. 995–1001.

[42] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[43] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *International Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, March 2002.

[44] J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue, "Motion planning for humanoid robots," in *In Proc. 20th Int'l Symp. Robotics Research (ISRR'03)*, 2003.

[45] E. Ferre and J.-P. Laumond, "An iterative diffusion algorithm for part disassembly," in *IEEE International Conference on Robotics and Automation*, New Orleans, Louisiana, April 2004, pp. 3149–3154.

[46] E. Plaku, K. Bekris, B. Chen, A. Ladd, and L. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," *IEEE Transactions on Robotics and Automation*, vol. 21, no. 4, pp. 597–608, Aug. 2005.

[47] M. Morales, L. Tapia, R. Pearce, S. Rodriguez, and N. M. Amato, "A machine learning approach for feature-sensitive motion planning," Texas A&M University, Tech. Rep., 2004.

[48] S. Rodriguez, S. Thomas, R. Pearce, and N. M. Amato, "RESAMPL: A region-sensitive adaptive motion planner," in *Algorithmic Foundation of Robotics VII*. Springer, STAR 47, 2008, pp. 285–300.

[49] L. Jaillet, A. Yershova, S. M. LaValle, and T. Siméon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs," in *International Conference on Intelligent Robots and Systems*, Edmonton, Canada, August 2005, pp. 2851–2856.

[50] A. Yershova, L. Jaillet, T. Siméon, and S. M. LaValle, "Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain," in *IEEE International Conference on Robotics and Automation*, Barcelona, Spain, April 2005, pp. 3856–3861.

[51] A. M. Ladd and L. E. Kavraki, "Motion planning in the presence of drift, underactuation and discrete system changes," in *Robotics: Science and Systems*, Boston, Massachusetts, June 2005, pp. 233–241.

[52] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with RRTs," in *IEEE International Conference on Robotics and Automation*, Orlando, Florida, May 2006, pp. 1243–1248.

[53] J. P. v. d. Berg and M. H. Overmars, "Path planning in repetitive environments," in *Methods and Models in Automation and Robotics*, 2006, pp. 657–662.

[54] B. Burns and O. Brock, "Single-query motion planning with utility-guided random trees," in *IEEE International Conference on Robotics and Automation*, Rome, Italy, April 2007, pp. 3307–3312.

[55] K. E. Bekris and L. E. Kavraki, "Greedy but safe replanning under kinodynamic constraints," in *IEEE International Conference on Robotics and Automation*, Rome, Italy, April 2007, pp. 704–710.

[56] E. Plaku, M. Y. Vardi, and L. E. Kavraki, "Discrete search leading continuous exploration for kinodynamic motion planning," in *Robotics: Science and Systems*, W. Burgard, O. Brock, and C. Stachniss, Eds. Atlanta, Georgia: MIT Press, June 2007, pp. 326–333.

[57] I. A. Şucan and L. E. Kavraki, "On the implementation of single-query sampling-based motion planners," in *IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, May 2010, pp. 2005–2011.

[58] F. Lamiraux, E. Ferr, and E. Vallee, "Kinodynamic motion planning: Connecting exploration trees using trajectory optimization methods," in *IEEE International Conference on Robotics and Automation*, vol. 4, New Orleans, Louisiana, April 2004, pp. 3987–3992.

[59] P. Cheng, E. Frazzoli, and S. M. LaValle, "Improving the performance of sampling-based planners by using a symmetry-exploiting gap reduction algorithm," in *IEEE International Conference on Robotics and Automation*, vol. 5, New Orleans, Louisiana, April 2004, pp. 4362–4368.

[60] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Computer Science Dept., Iowa State University, Tech. Rep. 11, 1998.

[61] E. Plaku, L. E. Kavraki, and M. Y. Vardi, "Motion planning with dynamics by a synergistic combination of layers of planning," *IEEE Transactions on Robotics*, vol. 26, pp. 469–482, June 2010.

[62] A. M. Ladd and L. E. Kavraki, "Fast tree-based exploration of state space for robots with dynamics," in *Algorithmic Foundations of Robotics VI.* Springer, STAR 17, 2005, pp. 297–312.

[63] A. Shkolnik, M. Walter, and R. Tedrake, "Reachability-guided sampling for planning under differential constraints," in *IEEE International Conference on Robotics and Automation*, Kobe, Japan, May 2009, pp. 2859–2865.

[64] A. M. Ladd, "Direct motion planning over simulation of rigid body dynamics with contact," Ph.D. dissertation, Rice University, Houston, Texas, December 2006.

[65] P. Cheng and S. M. LaValle, "Reducing metric sensitivity in randomized trajectory design," in *IEEE International Conference on Robotics and Automation*, Seoul, Korea, May 2001, pp. 43–48.

[66] M. Kalisiak and M. v. d. Panne, "RRT-blossom: RRT with a local flood-fill behavior," in *IEEE International Conference on Robotics and Automation*, Orlando, Florida, May 2006, pp. 1237–1242.

[67] H. Kurniawati and D. Hsu, "Workspace importance sampling for probabilistic roadmap planning," in *International Conference on Intelligent Robots and Systems*, Sendai, Japan, September 2004, p. 1618âĂŞ1623.

[68] S. Dalibard and J.-P. Laumond, "Control of probabilistic diffusion in motion planning," in *Algorithmic Foundations of Robotics VIII.* Springer, STAR 57, 2009, pp. 467–481.

[69] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice.* Morgan Kaufmann Publishers, 2004.

[70] A. Miller and P. K. Allen, "GraspIt!: a versatile simulator for robotic grasping," *IEEE Robotics and Automation Magazine*, vol. 11, no. 4, 2004.

[71] M. Ciocârlie, K. Hsiao, E. G. Jones, S. Chitta, R. Rusu, and I. Şucan, "Towards reliable grasping and manipulation in household environments," in *International Symposium on Experimental Robotics*, 2010.

[72] R. Alami, J.-P. Laumond, and T. Siméon, "Two manipulation planning algorithms," in *International Workshop on the Algorithmic Foundations of Robotics*, 1994.

[73] C. L. Nielsen and L. E. Kavraki, "A two level Fuzzy PRM for manipulation planning," in *International Conference on Intelligent Robots and Systems*, Takamatsu, Japan, 2000, pp. 1716–1722.

[74] R. Alami, T. Siméon, and J.-P. Laumond, "A geometrical approach to planning manipulation tasks," in *International Symposium on Robotics Research*, 1989, pp. 113–119.

[75] T.-Y. Li and J.-C. Latombe, "On-line manipulation planning for two robot arms in a dynamic environment," *International Journal of Robotics Research*, vol. 16, pp. 144–167, 1997.

[76] Y. Koga and J.-C. Latombe, "On multi-arm manipulation planning," in *IEEE International Conference on Robotics and Automation*, San Diego, California, May 1994, pp. 945–952.

[77] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *International Journal of Robotics Research*, vol. 17, pp. 315–337, 1998.

[78] E. Plaku and G. D. Hager, "Sampling-based motion and symbolic action planning with geometric and differential constraints," in *IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, May 2010, pp. 5002–5008.

[79] A. Bhatia, L. E. Kavraki, and M. Y. Vardi, "Sampling-based motion planning with temporal goals," in *IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, May 2010, pp. 2689–2696.

[80] J. Chestnutt, M. Lau, K. M. Cheung, J. Kuffner, J. K. Hodgins, and T. Kanade, "Footstep planning for the Honda ASIMO Humanoid," in *IEEE International Conference on Robotics and Automation*, Barcelona, Spain, April 2005.

[81] K. Hauser, T. Bretl, J.-C. Latombe, and B. Wilcox, "Motion planning for a six-legged lunar robot," in *International Workshop on the Algorithmic Foundations of Robotics*, New York City, July 2006.

[82] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for dynamic robots," *Automatica*, vol. 45, pp. 343–352, 2009.

[83] I. A. Şucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Algorithmic Foundation of Robotics VIII*. Springer, STAR 57, 2009, pp. 449–464.

[84] S. Rodriguez, S. Thomas, R. Pearce, and N. M. Amato, "RESAMPL: A region-sensitive adaptive motion planner," in *International Workshop on the Algorithmic Foundations of Robotics*, New York City, July 2006.

[85] I. A. Şucan, J. F. Kruse, M. Yim, and L. E. Kavraki, "Reconfiguration for modular robots using kinodynamic motion planning," in *ASME Dynamic Systems and Control Conference*, Michigan, Ann Arbor, October 2008.

[86] R. Diankov, N. Ratliff, D. Ferguson, S. Srinivasa, and J. J. Kuffner., "Bispace planning: Concurrent multi-space exploration," in *Robotics: Science and Systems*, Zurich, Switzerland, June 2008.

[87] I. A. Şucan and L. E. Kavraki, "A sampling-based tree planner for systems with complex dynamics," *IEEE Transactions on Robotics*, vol. 27, no. 6, 2011.

[88] I. A. Şucan, "Kinodynamic motion planning for high-dimensional physical systems," Master's thesis, Rice University, November 2008.

[89] R. Tedrake, "LQR-trees: Feedback motion planning on sparse randomized trees," in *Robotics: Science and Systems*, Seattle, USA, June 2009.

[90] W. B. Johnson and J. Lindenstrauss, "Extensions of Lipschitz mappings into a Hilbert space," *Contemporary Mathematics*, vol. 26, pp. 189–206, 1984.

[91] I. A. Şucan and L. E. Kavraki, "On the performance of random linear projections for sampling-based motion planning," in *International Conference on Intelligent Robots and Systems*, St. Louis, Missouri, October 2009, pp. 2434–2439.

[92] J. Sastra, S. Chitta, and M. Yim, "Dynamic rolling for a modular loop robot," *International Journal of Robotics Research*, vol. 39, pp. 421–430, January 2008.

[93] "The Open Motion Planning Library." [Online]. Available: http://ompl.kavrakilab.org

[94] "Motion Strategy Library." [Online]. Available: http://msl.cs.uiuc.edu/msl/

[95] "Motion Planning Kit." [Online]. Available: http://robotics.stanford.edu/~mitul/mpk/

[96] A. V. Estrada, J. M. Lien, and N. M. Amato, "VIZMO++: a visualization, authoring, and educational tool for motion planning," in *Proc. 2006 IEEE Intl. Conf. on Robotics and Automation*, 2006, pp. 727–732.

[97] "Kineoworks." [Online]. Available: http://www.kineocam.com/kineoworks-library.php

[98] E. Plaku, K. E. Bekris, and L. E. Kavraki, "OOPS for motion planning: An online open-source programming system," in *Proc. 2007 IEEE Intl. Conf. on Robotics and Automation*, Rome, Italy, 2007, pp. 3711–3716.

[99] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010. [Online]. Available: http://www.programmingvision.com/rosen_diankov_thesis.pdf

[100] Boost. [Online]. Available: http://www.boost.org

[101] S. Redon, A. Kheddar, and S. Coquillart, "Fast continuous collision detection between rigid bodies," *Computer Graphics Forum*, vol. 21, no. 3, pp. 279–287, 2002.

[102] E. R. Johnson and T. D. Murphey, "Scalable variational integrators for constrained mechanical systems in generalized coordinates," *IEEE Trans. on Robotics*, vol. 25, no. 6, pp. 1249 –1261, Dec. 2009. [Online]. Available: http://trep.sourceforge.net

[103] N. Amato, O. Bayazit, L. Dale, C. Jones, and D. Vallejo, "OBPRM: An obstacle-based PRM for 3D workspaces," in *Robotics: The Algorithmic Perspective*, P. K. Agarwal, L. E. Kavraki, and M. T. Mason, Eds. A.K. Peters, 1999, pp. 155–168.

[104] V. Boor, M. H. Overmars, and A. F. van der Stappen, "The Gaussian sampling strategy for probabilistic roadmap planners," in *Proc. 1999 IEEE Intl. Conf. on Robotics and Automation*, vol. 2, 1999, pp. 1018–1023.

[105] I. A. Şucan, M. Kalakrishnan, and S. Chitta, "Combining planning techniques for manipulation using realtime perception," in *IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, May 2010, pp. 2895–2901.

[106] I. A. Şucan and L. E. Kavraki, "Mobile manipulation: Encoding motion planning options using task motion multigraphs," in *IEEE International Conference on Robotics and Automation*, Shanghai, China, May 2011, pp. 5492–5498.

[107] ——, "On the advantages of task motion multigraphs for efficient mobile manipulation," in *International Conference on Intelligent Robots and Systems*, 2011, (accepted).

[108] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. Wiley, 2005.

[109] J. M. Ahuactzin and K. Gupta, "The kinematic roadmap: a motion planning based global approach for inverse kinematics of redundant robots," *IEEE Transactions on Robotics and Automation*, vol. 15, pp. 653–669, 1999.

[110] L. Sentis and O. Khatib, "A whole-body control framework for humanoids operating in human environments," in *IEEE International Conference on Robotics and Automation*, Orlando, Florida, May 2006, pp. 2641–2648.

[111] P. E. Missiuro and N. Roy, "Adapting probabilistic roadmaps to handle uncertain maps," in *IEEE International Conference on Robotics and Automation*, Orlando, Florida, May 2006, pp. 1261–1267.

[112] Y. Huang and K. Gupta, "Collision-probability constrained PRM for a manipulator with base pose uncertainty," in *International Conference on Intelligent Robots and Systems*, St. Louis, Missouri, October 2009, pp. 1426–1432.

[113] R. Alterovitz, T. Siméon, and K. Goldberg, "The stochastic motion roadmap: A sampling framework for planning with Markov motion uncertainty," in *Robotics: Science and Systems*, W. Burgard, O. Brock, and C. Stachniss, Eds. Atlanta, Georgia: MIT Press, June 2007.

[114] S. Chakravorty and S. Kumar, "Generalized sampling based motion planners with application to nonholonomic systems," in *IEEE International Conference on Systems, Man and Cybernetics*, October 2009, pp. 4077–4082.

[115] J. van den Berg, S. Patil, R. Alterovitz, P. Abbeel, and K. Goldberg, "LQG-based planning, sensing, and control of steerable needles," in *International Workshop on the Algorithmic Foundations of Robotics*, Singapore, December 2010.

[116] M. L. Puterman, *Markov Decision Processes*, 2nd ed. Wiley, 2005.

[117] S. Prentice and N. Roy, "The belief roadmap: Efficient planning in linear POMDPs by factoring the covariance," *International Journal of Robotics Research*, vol. 28, pp. 1448–1465, 2009.

[118] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, pp. 99–134, 1998.